НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет прикладної математики
Кафедра системного програмування
і спеціалізованих комп'ютерних систем

«На правах рукопису»
УДК 004.8

До захисту допущено:
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«__»_____2025р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Системне програмування і спеціалізовані комп'ютерні системи»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Метод генерації опису програмного коду з використанням моделі штучного інтелекту»

Виконала:
студентка ІІ курсу, групи КВ-41мп
Костюченко Альбіна Валентинівна _____

Науковий керівник:
Доцент кафедри СПСКС, к.т.н., доцент,
Петрашенко Андрій Васильович _____

Рецензент:
Доцент кафедри ПЗКС, к.т.н., доцент
Заболотня Тетяна Миколаївна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра системного програмування

і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп‘ютерна інженерія»

Освітньо-професійна програма **«Системне програмування та спеціалізовані комп‘ютерні системи»**

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____2025 р

ЗАВДАННЯ
на магістерську дисертацію студентці

Костюченко Альбіні Валентинівні

1. Тема дисертації «Метод генерації опису програмного коду з використанням моделі штучного інтелекту», науковий керівник дисертації Петрашенко Андрій Васильович, к.т.н., доцент, затверджені наказом по університету від «6» листопада 2025 р. №4836-с.

2. Термін подання студентом дисертації 16 грудня 2025

3. Об’єкт дослідження: процес генерації опису коду.

4. Предмет дослідження: попередньо навчені моделі штучного інтелекту з донавчанням на специфічних наборах даних для виконання задач програмування та документування.

5. Перелік завдань, які потрібно розробити:

1) проаналізувати існуючі програми для автоматичного генерування документації коду з метою визначення проблем, що потребують вдосконалення;
2) опрацювати дослідження та публікації за темою;
3) визначити вимоги до системи генерації опису коду;
4) розробити архітектуру створюваної системи;

5) розробити структурну та функціональну схему системи;
6) розробити прототип та інтерфейс системи;
7) протестувати систему та провести аналіз отриманих результатів;
8) сформулювати висновки.

6. Орієнтовний перелік ілюстративного матеріалу: презентація.

7. Орієнтовний перелік публікацій:

1) Костюченко А. В., Петрашенко А. В. Метод генерації опису програмного коду з використанням моделі штучного інтелекту. Прикладна математика та комп'ютинг ПМК 2025 : збірник тез доповідей XVIII наук.-практ. конференції магістрантів та аспірантів (м. Київ, 28-30 лист. 2023 р.).

2) A. Kostiuchenko, O. Mukhanova, A. Petrashenko. Method for generating source code description using an artificial intelligence model. Збірник доповідей International R&D Online Student Conference And Competition "Science And Technology Of The XXI Century (м. Київ, 4 груд. 2025 р.).

8. Дата видачі завдання: 25 жовтня 2025

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Вивчення літератури за тематикою МД	11.12.2024	
2.	Розроблення та узгодження технічного завдання	04.09.2025	
3.	Аналіз існуючих рішень	15.09.2025	
4.	Підготовка матеріалів розділів МД	10.10.2025	
5.	Розробка апаратного та програмного забезпечення системи	10.11.2025	
6.	Тестування розробленого апаратного та програмного забезпечення системи	20.11.2025	
7.	Оформлення документації МД	30.11.2025	
8.	Попередній розгляд магістерської дисертації на кафедрі	05.12.2025	

Студент _____ Альбіна КОСТЮЧЕНКО

Науковий керівник _____ Андрій ПЕТРАШЕНКО

РЕФЕРАТ

Актуальність теми: Тема є актуальною, оскільки у даний час існує багато великих проектів, які розробляються протягом тривалого періоду часу та потребують підтримки та розуміння коду без пояснень. Швидкий розвиток технологій та необхідність постійної розробки нових функцій і підтримки вже існуючих потребує постійного оновлення документації. Написання хорошої документації є цінною навичкою, що потребує досвіду, концентрації та розуміння структури проекту. Як наслідок, велика кількість розробників вважають процес написання документації важким і думають, що час, витрачений на це, можна було б використати більш продуктивно. Саме тому є попит на сервіси, які допомогають автоматизувати цей процес.

Об’єкт дослідження: Об'єктом дослідження є процес генерації опису коду.

Предмет дослідження: Предметом дослідження є попередньо навчені моделі штучного інтелекту з донавчанням на специфічних наборах даних для виконання задач програмування та документування.

Мета роботи: Метою даної роботи є підвищення ефективності автоматизованої генерації програмної документації. У рамках виконання даного завдання було опрацьовано необхідний теоретичний матеріал, вивчено уже існуючі рішення даної проблеми та розроблено і реалізовано власний новий спосіб генерування опису програмного коду, який більш точно визначав призначення фрагментів коду, чітко розумів структуру та залежності між його складовими.

Методи дослідження: Дослідження базується на аналізі літератури, статистичних методах, а також методах машинного навчання та інтелектуального аналізу даних. Зокрема було використано методи синтаксичного аналізу коду та побудови абстрактного синтаксичного дерева (AST), метод формування навчального корпусу, методи навчання та донавчання трансформерних та графових моделей. Для оцінки переваг

донавченої моделі було застосовано метод порівняльного моделювання та автоматизованої оцінки якості тексту (у даному випадку BERTScore).

Наукова новизна: У рамках дослідження було:

1. Розроблено більш точний метод генерації опису програмного коду.
2. Донавчено мовну модель типу T5 на наборах коду, зібраного з сервісу GitHub, для виконання нею специфічних задач, пов’язаних з описом коду.
3. Протестовано та проаналізовано результати роботи моделі порівняно їх з результатами, отриманими від уже існуючого аналогу. Як наслідок, донавчена модель показує кращі результати та генерує більш точні коментарі, ніж базова T5.
4. Створено та навчено на власному наборі даних модель типу GNN, для побудови графів залежності частин програми та їхньої подальшої візуалізації. Врахування структури проекту та побудова AST коду дає порівняно кращий результат під час коментування коду.
5. Жоден з розглянутих аналогів не використовує одночасно моделі T5 та GNN і генератор синтаксичних аналізаторів ANTLR для покращення якості документації, також таких підхід не згадується в опрацьованих дослідженнях. Тому сама структура запропонованої системи є унікальною.

Практична цінність: Результати роботи можуть бути використані для створення документації для програмного забезпечення, створеного за допомогою мови Python, яка на даний момент є широко використовуваною в майже усіх сферах інформаційних технологій. Можливе донавчання моделі для роботи з іншими високорівневими мовами програмування.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XVIII науковій конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2025 та на II Міжнародній підсумковій науково-практичній онлайн-

конференції II Міжнародного конкурсу студентських наукових робіт з англійської мови “Advances in Science and Technology”.

Структура та обсяг роботи: робота складається з реферату, змісту, вступу, чотирьох розділів та висновку.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень.

У першому розділі розглянуто існуючі аналоги, який дає змогу визначити основні переваги та недоліки цих рішень, а також оглянуто дослідження за темою роботи.

У *другому розділі* наведено перелік існуючих інструментів, які доступні для реалізації дослідження, їхній огляд. На основі даних у першому та другому розділах запропоновано власний підхід, викладений у висновках до розділу.

У третьому розділі наведено особливості запропонованого методу та його реалізації, зокрема деталі процесу навчання моделей штучного інтелекту та роль ембедингів у проекті.

У *четвертому розділі* наведено дані про структуру розробленої системи, результати дослідження, тестові приклади та порівняння власних результатів з аналогами.

У висновках представлені результати проведеної роботи, огляд виконаних задач та перспектив для подальшого розвитку.

Ключові слова: машинне навчання, T5, GNN, генератор опису коду, навчання моделі, обробка природної мови, документація, AST.

ABSTRACT

Relevance of the topic. The topic is relevant, since currently there are many large projects that are developed over a long period of time and require support and understanding of the code without explanations. The rapid development of technologies and the need to constantly develop new features and support existing ones requires constant updating of documentation. Writing good documentation is a valuable skill that requires experience, concentration and understanding of the project structure. As a result, a large number of developers consider the process of writing documentation difficult and think that the time spent on it could be used more productively. That is why there is a demand for services that help automate this process.

Object of research. The object of research is the process of generating code descriptions.

Object of research. The object of research is pre-trained artificial intelligence models with additional training on specific data sets for performing programming and documentation tasks.

Purpose of work. The purpose of this work is to increase the efficiency of automated generation of software documentation. As part of this task, the necessary theoretical material was worked out, existing solutions to this problem were studied, and a new method of generating a description of the program code was developed and implemented, which more accurately determined the purpose of code fragments, clearly understood the structure and dependencies between its components.

Research methods. The study is based on literature analysis, statistical methods, as well as machine learning and data mining methods. In particular, methods of syntactic code analysis and construction of an abstract syntax tree (AST), the method of forming a training corpus, methods of training and retraining of transformer and graph models were used. To assess the advantages of the retrained model, the method of comparative modeling and automated text quality assessment (in this case, BERTScore) was used.

Scientific novelty. As part of the study, the following were done:

1. A more accurate method of generating a description of the program code was developed.
2. A language model of the T5 type was retrained on sets of code collected from the GitHub service to perform specific tasks related to code description.
3. The results of the model were tested and analyzed, comparing them with the results obtained from an existing analogue. As a result, the retrained model shows better results and generates more accurate comments than the basic T5.
4. A GNN-type model was created and trained on its own dataset to construct dependency graphs of program parts and their subsequent visualization. Taking into account the project structure and building an AST code gives a relatively better result when commenting on the code.
5. None of the considered analogues simultaneously uses the T5 and GNN models and the ANTLR parser generator to improve the quality of documentation, and such an approach is not mentioned in the developed studies. Therefore, the very structure of the proposed system is unique.

Practical value. The results of the work can be used to create documentation for software created using the Python language, which is currently widely used in almost all areas of information technology. It is possible to further train the model to work with other high-level programming languages.

Approbation of the work. The main provisions and results of the work were presented and discussed at the XVIII Scientific Conference of Master's and PhD Students "Applied Mathematics and Computing" PMK-2025 and at the II International Final Scientific and Practical Online Conference of the II International Competition of Student Scientific Papers in English "Advances in Science and Technology".

Structure and scope of the work. The work consists of an abstract, table of contents, introduction, four sections and a conclusion.

The *introduction* provides a general description of the work, an assessment of the current state of the problem is made, the relevance of the research direction is substantiated, the goal and objectives of the research are formulated.

The *first section* considers existing analogues, which allows us to determine the main advantages and disadvantages of these solutions, and also reviews research on the topic of the work.

The *second section* provides a list of existing tools that are available for the implementation of the study, their review. Based on the data in the first and second sections, our own approach is proposed, which is set out in the conclusions to the section.

The *third section* presents the features of the proposed method and its implementation, in particular, details of the process of training artificial intelligence models and the role of embeddings in the project.

The *fourth section* provides data on the structure of the developed system, research results, test cases and comparison of our own results with analogues.

The *conclusions* present the results of the work, a review of the tasks performed and prospects for further development.

Keywords: machine learning, T5, GNN, code description generator, model training, natural language processing, documentation, AST.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Нейронна мережа – здібна до навчання математична модель, котра імітує роботу нервової системи живих організмів. Наразі широко застосовуються в задачах розпізнавання, прогнозування та класифікації.

JSON (JavaScript Object Notation) – текстовий формат для обміну даними, який являє собою впорядкований набір значень або частіше набір пар «ключ-значення», забезпечує обмін даними між серверами або між браузером та сервером. Виник із мови JavaScript, але наразі може використовуватись з багатьма мовами програмування.

Фреймворк – платформа, яка визначає структуру розроблюваного проєкту. Фреймворк полегшує як розробку окремих компонентів, так і їхнє об’єднання. На відміну від бібліотеки, фреймворк накладає певні обмеження на програмний код та формує каркас додатку, який потім розширюється згідно з потребами.

Python – високорівнева мультипарадигмальна мова програмування; широко застосовується у веброзробці, машинному навчанні, математичних обчисленнях, роботі з базами даних, створенні ігор, тощо.

JavaScript (JS) – високорівнева об'єктно-орієнтована мова програмування з C-подібним синтаксисом, є реалізацією стандарту ECMAScript; на сьогодні мова набула найбільшого поширення у веб-розробці, але в останні роки все частіше використовується поза браузером.

Graph Neural Networks (GNN) – тип нейронних мереж, що використовуються для обробки даних, які можна представити у вигляді графів.

Abstract Syntax Tree (AST) – орієнтоване дерево, де гілками є оператори мови програмування, а листками – операнди. Створюється для аналізу та модифікації коду.

ВСТУП

У наш час сфера інформаційних технологій розвивається надшвидко. На даний момент Github, найбільший у світі веб-сервіс для зберігання та публікації коду, налічує більше 47 мільйонів публічних репозиторіїв [1]. Проте, на жаль, не весь код, що існує, є зрозумілим та має чіткі та розгорнуті пояснення у вигляді коментарів, ще рідше зустрічається повноцінна, чітко структурована документація. І це є досить серйозною проблемою, оскільки над великими проектами часто працює не одна сотня розробників, котрі час від часу змінюються. Відсутність якісної документації підвищує кількість часу, необхідного для розуміння та редагування коду.

Саме тому існує програмне забезпечення, яке генерує документацію на основі коментарів у коді, назв та функціональності класів та методів або загального контексту. Певні програми створюють опис коду за шаблонами, інші використовують обробку природної мови для аналізу залежностей та функцій, які виконують ті чи інші частини коду.

Найстаріша система для документування коду була створена ще у 1997 році, проте стрімко розвиватись ця галузь програмного забезпечення почала лише двадцять років потому – з появою моделей-трансформерів. Ті рішення, які уже існують, постійно вдосконалюються, і щоразу знаходяться нові.

Метою даної роботи є розробка та реалізація власного нового методу генерування опису програмного коду, який би більш точно визначав функції фрагментів коду, чітко розумів структуру проекту та залежності між його складовими. Для виконання роботи було поставлено такі задачі:

1. Вивчення літератури та довідкових матеріалів за темою роботи, а також вивчення вже існуючих рішень та методів, які вони використовують.
2. Ознайомлення з типами існуючих моделей машинного навчання, їхніми перевагами, недоліками і сферами застосування.
3. Вивчення доступних інструментів, які можна використати для реалізації власної системи. Детальне ознайомлення з бібліотеками

для роботи з моделями машинного навчання, аналізу коду, створення графів залежності та генерації вихідного документа.

4. Створення власного методу вирішення поставленої задачі, встановлення вимог до створюваної системи, збір необхідних даних для навчання моделі.
5. Проектування створюваної системи, написання програмного коду та тестування.
6. Аналіз отриманих результатів, підбиття підсумків та написання висновків.

Перелік даних завдань охоплює всі необхідні етапи дослідження. Результати їхнього послідовного виконання будуть приведені у даній роботі.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ В ГАЛУЗІ ГЕНЕРАЦІЇ ОПИСУ ПРОГРАМНОГО КОДУ

1.1. Генератори статистичного опису коду на основі шаблонів

1.1.1. Doxygen

Doxygen – це система документування вихідних текстів, яка була вперше випущена 26 жовтня 1997 року [2] і на даний момент підтримує десять мов програмування, серед яких PHP, Java, C, Fortran і VHDL. Система підтримує генерацію документації у форматах HTML, XML, LaTeX, RTF та man. Doxygen досить широко застосовується і використовується у таких проектах як Torque Game Engine, AbiWord, Mozilla, Crystal Space та інших.

Doxygen складається з кількох компонентів [3], таких як C-препроцесор, парсер конфігурації, парсер мови, органайзер даних, генератор вихідного файлу, тощо (рис. 1.1).

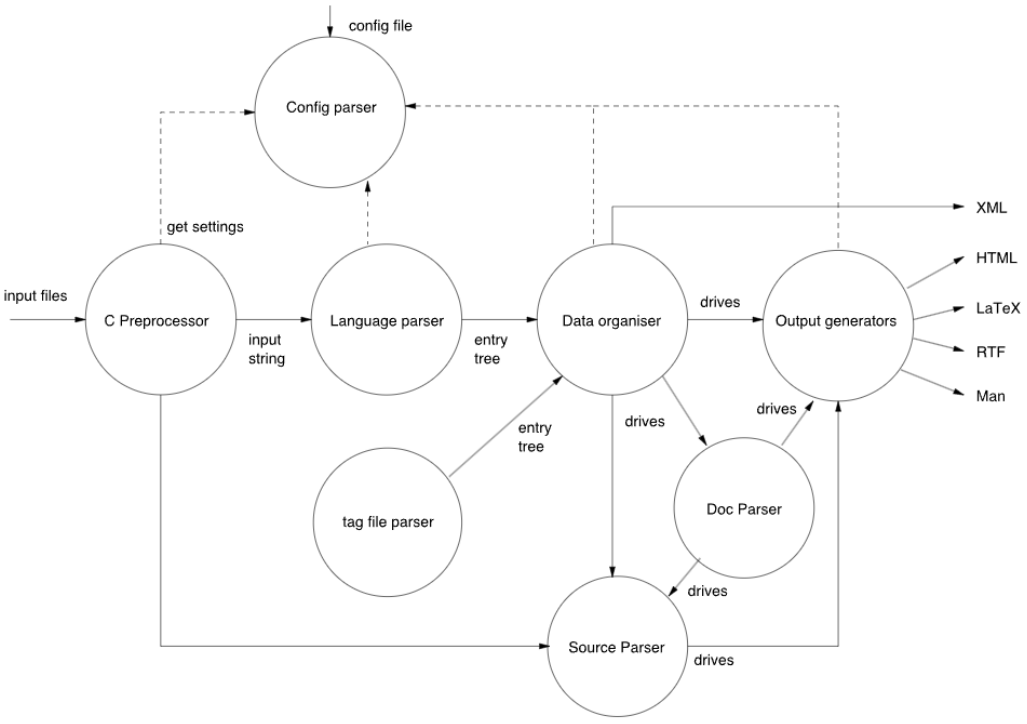

Рис. 1.1 – Схема потоків даних у Doxygen

В першу чергу аналізується файл конфігурації, котрий керує налаштуваннями проекту. Отримані налаштування зберігаються у файлі src/config.h у класі Config. Сам парсер, написаний за допомогою flex (Fast Lexical Analyzer Generator) – генератора лексичних аналізаторів, розбиває текст на окремі токени за допомогою вказаних регулярних виразів. Парсер находиться у окремій бібліотеці src/config.l.

Кожен параметр конфігурації має тип String, List, Enum, Int або Bool. Їхні значення глобально доступні через функції типу Config_getXXX(), де XXX – тип параметра. Наприклад, функція Config_getBool (GENERATE_TESTLIST) повертає вказівник на булеве значення True, якщо testlist доступний у файлі конфігурації.

Далі всі файли з вхідним кодом, які були вказані у конфігураційному файлі, за замовчуванням проходять обробку спеціалізованим препроцесором C, який створений за допомогою того ж flex і має кілька відмінностей від стандартного та знаходиться у файлі src/pre.l. По-перше, препроцесор за замовчуванням не розширює виклики макросів, хоча це можна налаштувати, по-друге – команди #include аналізуються, але вказані там файли не підключаються. Кожен файл викликається функцією preprocessFile(), а результат попередньої обробки додається до буферу символів у форматі «0x06 file name N; 0x06 preprocessed contents of file N».

Отримані файли обробляє аналізатор мови (файл src/scanner.l), який працює як кінцевий автомат з використанням flex. Задача аналізатора – це перетворення отриманого на попередньому кроці буфера символів у абстрактне синтаксичне дерево. Для всіх мов працює один великий сканер, що сповільнює роботу програми.

Після цього створюються словники вилучених класів, файлів, просторів імен, змінних, функцій, пакетів, сторінок і груп. Окрім створення словників, під час цього кроку обчислюються відносини (наприклад, відносини спадкування) між отриманими сутностями. Наявні у коді блоки коментарів зберігаються як рядки в сутностях, які вони документують.

Якщо увімкнено перегляд вихідного коду або якщо в документації зустрічаються фрагменти коду, запускається аналізатор вихідного коду. Синтаксичний аналізатор коду намагається зробити перехресне посилання на вихідний код, який він аналізує, із задокументованими сутностями. Він також виконує підсвічування синтаксису джерел. Вихідні дані записуються безпосередньо до вихідних генераторів. Основна точка входу для аналізатора коду parseCode()оголошена в src/code.h.

Після збору даних і перехресних посилань програма оформлює вихідні дані у різних форматах. Для цього використовуються методи, надані абстрактним класом OutputGenerator. XML-дані генеруються безпосередньо із зібраних структур даних. У майбутньому розробники планують використовувати XML як проміжну мову. Перевага наявності проміжної мови у тому, що незалежно розроблені інструменти, написані різними мовами, зможуть витягувати інформацію з вихідних даних XML.

Окрім цього, Doxygen має невеликий набір тестів для перевірки цілісності коду. Тести можна запускати за допомогою команди «make tests». Якщо потрібно лише один або кілька тестів, можна встановити змінну TEST_FLAGS під час виконання тесту, наприклад «make TEST_FLAGS="--id 5" tests», або для кількох тестів «make TEST_FLAGS="--id 5 --id 7" tests».

1.1.2. MkDocs

MkDocs створює статичні сайти з документацією, отриманою із текстових файлів типу Markdown [4]. Генератор має вбудовану тему Material for MkDocs та підтримує велику кількість плагінів. Наприклад, можна підключити автоматичну генерацію таблиць, інтеграцію з системою контролю версій та підтримку діаграм.

MkDocs написаний за допомогою мови Python та розповсюджується як пакет для Python. Основні налаштування сайту описуються у файлі

mkdocs.yml, де вказується назва сайту, навігація, підключені плагіни, тощо. Всі файли організовуються на основі файлів Markdown та дерева директорій. Кожен файл Markdown додається у створену директорію docs/ та перетворюється на окрему HTML-сторінку сайту, яка складається з шаблонів на основі Jinja2. Проект створюється командою mkdocs new, статичний сайт створюється командою mkdocs build. Після цього всі HTML-файли зберігаються у папці site/, яка доступна на локальному хості після виконання команди mkdocs serve та може бути завантажена на будь-який хостинг.

1.1.3. Swagger

Swagger – це інструмент для створення документації до API-сервісів. Він створює окрему сторінку з інтерактивною документацією, де відповідно до шаблону описано кожну із задокументованих функцій з прикладами використання та зразками вхідих даних, які функція отримує або створює. Такий підхід дозволяє зручно протестувати розроблений API одразу у браузері.

Інструкції для Swagger можна записувати у форматі JSON або YAML [5]. Структура опису API відповідає стандартам OAS (OpenAPI Specification). Зокрема можна вказати специфікацію Swagger, назву, короткий опис та версію створеного API, окремі кінцеві точки (шляхи) та методи (операції) HTTP, які підтримуються цими кінцевими точками, методи аутентифікації, тощо.

Серверні елементи Swagger, такі як Swagger Codegen, Swagger Parser, написані мовою Java, інтерактивна документація Swagger UI створюється за допомогою мов Javascript, HTML та CSS. Swagger Editor, який дозволяє перевіряти специфікації на відповідність стандартам OpenAPI Specification та тестувати API у браузері, створений за допомогою ReactJS. Також для різних мов та фреймворків існують бібліотеки Swagger Annotations, Flask-

Swagger, FastAPI та інші, які дозволяють додавати анотації у програмний код.

Далі анотації після валідації автоматично перетворюються у OAS-файл. Swagger-сервери аналізують вхідний код, використовуючи статичний аналіз та рефлексію, щоб отримати інформацію про API. Потім всі зібрані дані перетворюються у інтерактивний односторінковий додаток, що складається з шаблонів. При цьому використовуються бібліотеки для рендерингу, такі як Handlebars.

1.2.Генератори опису коду з урахуванням контексту

1.2.1. GitHub Copilot

GitHub Copilot – це заснований на штучному інтелекті плагін, котрий вбудовується у популярні середовища розробки на зразок Visual Studio Code, IntelliJ IDEA, Neovim, тощо [6]. Він пропонує автозакінчення та навіть повноцінні блоки коду на основі кількох рядків програми або коментарів. Також Copilot може за потреби формувати коментарі для вже написаного коду.

Плагін створений за допомогою адаптації моделі-трансформера GPT-3 під назвою OpenAI Codex, котра була навчена на загальнодоступних даних, такі як публічні репозиторії та документація. Модель не запам’ятовує всі проекти користувача, але постійно вдосконалюється за рахунок нової доступної інформації із загального доступу. Також модель навчається на загальній статистиці, проте не використовує дані користувача напряму. Під час контекстного аналізу модель аналізує зміст файлу, локальні залежності та попередні рядки коду, а далі на основі цього пропонує коментарі або варіанти вирішення задачі.

Весь наявний контекст відправляється через редактор коду до API GitHub Copilot, там модель Codex аналізує його та пропонує кілька варіантів, які відправляються користувачу.

1.2.2. Swimm Auto-docs

Swimm Auto-docs – плагін, який працює у багатьох популярних середовищах розробки та редакторах коду [7]. Він аналізує файл повністю, за потреби враховує сусідні, розбиває їх на токени та формує рекомендації, які будуть підходити під стиль та структуру коду проекту.

Точно невідомо, яка саме модель застосовується в Swimm, скоріше за все це модель-трансформер на зразок Codex або GPT-3. Модель генерує абстрактні синтаксичні дерева, що дозволяє виділити важливі аспекти коду та точно описати взаємодію частин програми. Плагін тісно пов’язаний з системами контролю версій.

Підчас аналізу репозиторію, для якого потрібно сформувати документацію, Swimm Auto-docs автоматично виділяє основні компоненти проекту. Далі за допомогою штучного інтелекту генеруються описи методів та класів з детальним поясненням їхньої роботи. Всі пояснення зберігаються у Markdown-файлах [8].

1.2.3. Tabnine

Tabnine – це фактично плагін-аналог GitHub Copilot, який пропонує варіанти написання коду та анотації до нього у вигляді коментарів.

Розробники не розкривають всіх деталей функціонування інструменту. Відомо, що раніше Tabnine використовував модель GPT-2, але зараз в ньому використовується кастомізована, додатково донавчена модель, спеціалізована саме для задач написання коду [9]. Скоріше за все, це модель-трансформер типу T5, Codex або, найбільш ймовірно, GPT, яка навчена на наявному у вільному доступі коді.

Модель розуміє контекст коду, включно з викликами функцій, визначення змінних та коментарями, а потім генерує релевантні доповнення на основі зібраної інформації. Наприклад, якщо у коді

очікується виклик певного методу, Tabnine може запропонувати готовий метод разом з параметрами.

Якщо використовується локальна версія, інструмент може адаптуватися до проекту, навчаючись на приватній кодовій базі, але не відправляючи дані до хмарного сховища.

1.3. Огляд попередніх досліджень за темою

На одному з перших етапів дослідження було опрацьовано ряд публікацій, які так чи інакше розглядають використання моделей штучного інтелекту.

Зокрема у статті “Automatic comment generation for source code using external information by neural networks for computational thinking” [10] автори пропонують підхід до автоматичної генерації коментарів до вихідного коду, орієнтований на освітнє середовище та розвиток «комп’ютерного мислення». У статті вирішуються декілька фундаментальних проблем, що пов’язані із розумінням коду та побудовою зрозумілих коментарів, і для цього пропонуються конкретні архітектурні рішення. Дана робота є важливим дослідженням у галузі обробки коду нейронними мережами, оскільки вона розглядає не лише сам код, але й додатковий контекст, який супроводжує програму, що є суттєво новим.

Першою ключовою проблемою, на яку звертають увагу автори, є недостатність інформації, що міститься безпосередньо у вихідному коді. У багатьох випадках синтаксис або структура програми не дозволяють однозначно визначити її призначення, і без додаткового контексту модель не здатна коректно інтерпретувати логіку роботи. Для подолання цього обмеження дослідники вводять концепцію зовнішньої інформації, що включає формулювання задачі, опис вхідних і вихідних даних, текст навчального завдання та інші пояснювальні матеріали. Цей текст конкатенується з програмним кодом і подається на вхід моделі типу

encoder–decoder на базі LSTM. Таким чином, модель працює не лише з абстрактними синтаксичними структурами, а й з семантичним описом, що значно підвищує релевантність і змістовність згенерованих коментарів.

Автори багатьох попередніх робіт створювали програми, що генерують описи на рівні всієї функції чи файлу, однак іноді необхідно пояснювати роботу програми поетапно, на рівні окремих рядків або логічних блоків. Тому автори роблять акцент на деталізації коментарів і розробляють метод розбиття коду на структурні фрагменти – цикли, умовні оператори, декларації змінних та інші елементи. Для кожного такого блоку формується пара «код–коментар», що дозволяє навчати модель точковому поясненню дій програми, а не створенню узагальнених описів. Такий підхід робить результати моделі особливо корисними для початківців, оскільки пояснення стають локальними й прив’язаними до конкретних частин коду.

Ще однією задачею, яку автори вирішують у своїй роботі, є оцінювання якості згенерованих коментарів. Оскільки автоматичні метрики мають обмеження, дослідники використовують комбінований підхід. Для статистичного порівняння було застосовано метрики BLEU, ROUGE та METEOR, а для суб’єктивної оцінки зрозумілості, правильності та корисності отриманих пояснень залучили студентів і викладачів. Така двошарова система оцінювання забезпечує більш повне розуміння ефективності моделі в реальних освітніх сценаріях.

У технічному плані модель побудована на класичній архітектурі seq2seq з використанням LSTM в ролі енкодера та декодера, доповненої механізмом уваги. Це дозволяє декодеру фокусуватися на різних частинах коду під час генерації коментаря. Навчання моделі здійснюється за допомогою функції Cross-Entropy Loss, що є стандартним підходом у задачах генерування тексту. Об’єднання коду і зовнішнього тексту в один послідовний вхід дозволяє моделі будувати репрезентації, які враховують як синтаксичну, так і семантичну інформацію.

Таким чином, дана робота вирішує одразу кілька ключових проблем: нестачу семантичного контексту, відсутність деталізованих описів окремих блоків коду та складність об’єктивного оцінювання якості згенерованих коментарів, обмежуючись лише автоматичним аналізом і метриками.

Хоча запропонований підхід не враховує структурні залежності між модулями та не використовує абстрактні синтаксичні дерева чи графові моделі, він задає концептуальну основу, на якій можна будувати більш складні системи. У контексті даної магістерської роботи ця публікація служить важливим джерелом для порівняння, оскільки запропонована в дослідженні система може бути розширена за рахунок використання AST, графових нейронних мереж та сучасних моделей трансформерного типу, що дозволяє досягти вищої точності та структурної узгодженості згенерованої документації.

Наступним опрацьованим дослідженням є “Retrieve and Refine: Exemplar-based Neural Comment Generation” [11], у якому автори пропонують інноваційний підхід до автоматичної генерації коментарів для коду, який поєднує нейронні трансформерні моделі із механізмами пошуку найбільш релевантних прикладі у великій базі коментованого коду. На відміну від традиційних seq2seq-моделей, які генерують коментар безпосередньо з програмного коду, автори використовують двоступеневу архітектуру типу retrieve-and-refine, що дозволяє моделі використовувати вже існуючі якісні коментарі як відправну точку для побудови нового тексту.

Автори стверджують, що моделі мають обмежені можливості щодо генерації осмислених коментарів лише на основі вихідного коду. Особливо це помітно, якщо програма містить складні логічні конструкції, або використовує доменну термінологію, яку модель без прикладів іноді інтерпретує неправильно. Для подолання цього обмеження дослідники впроваджують компонент retrieve, який за допомогою подібності коду

знаходить приклад – існуючий фрагмент коду з якісним коментарем. Таким чином, система отримує не лише сам код, а й коментар, створений спеціалістом для схожої структури або логічного шаблону.

Другою важливою складовою системи є компонент refine, який дозволяє трансформеру змінювати, адаптувати та редагувати знайдений коментар під новий фрагмент коду. Автори застосовують модифіковану архітектуру типу енкодер–декодер, де енкодер отримує як сам код, так і коментар-прецедент. Декодер, у свою чергу, генерує новий коментар, поєднуючи інформацію з обох джерел. Тобто модель не створює текст з нуля, а відштовхується від уже наявного людського прикладу, що суттєво підвищує якість, змістовність та стильову природність згенерованих коментарів.

Третьою проблемою, на яку звертають увагу автори, є недостатня структурна адекватність коментарів, згенерованих чистими нейронними моделями. Підхід retrieve-and-refine використовує механізм стилістичного та семантичного наслідування: знайдений приклад надає релевантну лексику, структуру пояснення та термінологію. Завдяки цьому вихідний коментар має більше шансів бути зрозумілим, технічно точним і близьким до того, що пишуть реальні розробники.

Ще однією важливою частиною роботи є методика оцінювання моделі. Автори застосовують стандартні текстові метрики (BLEU, ROUGE), але також тестують модель на семантичних метриках, які краще відображають відповідність змісту, а не лише поверхневу схожість токенів. Крім того, вони проводять людську оцінку коментарів із залученням програмістів, які оцінюють такі параметри, як коректність, інформативність та відповідність коду. Результати демонструють, що підхід retrieve-and-refine суттєво перевершує моделі, які генерують текст без доступу до прецедентів.

З технічної точки зору архітектура моделі передбачає окремий етап обчислення схожості коду, для якого можуть використовуватися

багатовимірні векторні представлення, ембеддинги або моделі з попереднім навчанням.

Після того як найбільш схожі приклади відібрано, нейронна модель, заснована на трансформері, здійснює подальше уточнення, створюючи фінальний коментар. Навчання моделі виконується за допомогою крос-ентропійної функції втрат (Cross-Entropy Loss), що дозволяє покращити якість тексту на виході.

Що стосується графових нейронних мереж, ідея формування дерева проекту і його збереження у вигляді графа не є новою. Зокрема у доповіді «JGNN: Graph Neural Networks on Native Java» [12] автор описує, як перетворити програмний проект у граф залежностей.

Використовуються зв’язки імпорту, виклики функцій, структуру пакета та результати аналізу AST. Отриманий у результаті дослідження граф орієнтований і неоднорідний, оскільки містить кілька типів вузлів і зв'язків.

Розглянуто різні архітектури графованих нейронних мереж (GCN, GraphSAGE, GAT), націлювання ребер, детально пояснено механізм передачі повідомлень - властивість GNN, що дозволяє вузлу збирати інформацію від сусідів і розглядати контекст всього проекту.

Проте метою проекту було не документування коду, а пошук вразливостей. Створена модель навчена поділяти вузли графа на вразливі та безпечні. Було важливо, щоб GNN враховувала не тільки локальні характеристики файлу, але і його оточення: наявність небезпечних залежностей, з'єднань, маршрутів поширення даних. Це особливо актуально для сучасних проектів з великою кількістю зовнішніх бібліотек.

У статті показано, що класичні інструменти обмежуються лінійним або модульним аналізом коду, у той час як графові методи фіксують залежності від файлу до файлу глобально. Автор наводить приклади вразливостей, які точно визначені через інтермодульний контекст. З цією метою формується група програмних проектів з помітними вразливостями.

ВИСНОВКИ ДО РОЗДІЛУ 1

У цьому розділі було розглянуто уже існуючі рішення проблеми, описаної раніше. Було зібрано та проаналізовано інформацію про генератори статичної документації та генератори, які використовують технологію штучного інтелекту та враховують контекст.

Перший тип генераторів опису коду фактично збирає документацію із вже готових шаблонів, коментарів та назв змінних, класів та методів. Файли з вихідним кодом кілька разів обробляються парсером, що однозначно подовжує час роботи програм, а саме формування документації є менш гнучким, оскільки засновано на шаблонах. Окрім того, код, який документується, має вже бути добре структурованим та читабельним та бажано мати коментарі, адже від цього залежить якість отриманої документації. Деякі інструменти, такі як MkDocs, вимагають створення розробником окремих конфігураційних файлів або написання повноцінних JSON-анотацій, які потім збираються в односторінковий додаток, як у Swagger. Будь-які помилки у файлах конфігурації призводять до некоректного відображення документації або помилки її створення.

Генератори, котрі використовують для аналізу коду штучний інтелект, більш гнучкі та генерують більш розгорнуту та повну документацію для проектів, розробник може майже не втручатися в процес – все відбувається автоматично. Проте більшість таких інструментів працюють скоріше як штучні асистенти для написання коду, а створення опису коду є побічною функцією. Розробники таких генераторів не розкривають усіх деталей архітектури та принципів роботи моделей, які використовуються для аналізу коду, адже їхні алгоритми дають компанії конкурентну перевагу. Тому більшість інформації, яка знаходиться у відкритому доступі – це поради щодо користування розробленими інструментами, а не деталі внутрішньої структури.

З усієї зібраної інформації можна зробити висновок, що існує велика кількість інструментів, які можна використовувати для створення документації, проте вони мають певні недоліки і постійно удосконалюються.

Існуючі дослідження пропонують шляхи покращення результатів роботи моделей штучного інтелекту з кодом. Зокрема пропонується використання графових моделей для пошуку вразливостей у коді, а також використання моделей-трансформерів для написання коментарів із урахуванням структури коду та орієнтацією на еталонний коментар у процесі. Дані ідеї необхідно врахувати при розробці власного підходу.

2. ДОСЛІДЖЕННЯ, ВИБІР ТА ОГЛЯД ПІДХОДІВ ДО ВДОСКОНАЛЕННЯ ГЕНЕРАТОРІВ ОПИСУ ПРОГРАМНОГО КОДУ

2.1.Моделі штучного інтелекту для генерації опису коду

2.1.1. GPT

GPT або Generative Pre-trained Transformer – це група попередньо навчених моделей машинного навчання, які були створені компанією OpenAI [13]. Пізні версії GPT, наприклад, GPT-3, котра налічує 175 мільярдів параметрів, можна віднести до загального штучного інтелекту (AGI) – типу моделей, які за своїми здібностями наближені або рівні людському інтелекту. Модель працює з текстом, розбитим на окремі одиниці – токени.

GPT відносяться до типу моделей-трансформерів, які призначені для обробки послідовних даних і найбільш ефективні для роботи з текстом. Проте моделі GPT відрізняються від стандартних трансформерів тим, що використовує лише декодер, без кодерної частини трансформера [14]. Це зумовлено тим, що модель має враховувати попередній контекст, але їй не потрібно повністю кодувати всю вхідну інформацію. Такий підхід знижує обчислювальні витрати та спрощує архітектуру моделі.

Моделі GPT реалізують механізм self-attention (самоуваги), при якому алгоритм звертає увагу на попередні токени, враховує їхню значимість та намагається передбачити наступний токен. Наприклад, якщо у певному реченні зустрічається певне слово із ймовірністю 90%, то в окремо взятому випадку це слово там буде також із ймовірністю 90% і модель скоріш за все обере його. Для пошуку різних типів взаємозв'язку використовуються кілька «голів». Показник уваги вираховується за формулою (1):

$$Attention(Q, K, V) = Softmax\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) \cdot V \qquad (1)$$

де Q – матриця вагів запитів, K – матриця ключів, V – матриця значень токенів.

Також існує поняття температури, тобто параметра того, наскільки точною буде відповідь. Температура від одиниці і вище буде частіше давати креативні відповіді, проте є велика ймовірність генерації безглуздих результатів. Температура близько нуля забезпечує генерацію точних і консервативних відповідей.

Після навчання GPT може самостійно генерувати повноцінні тексти з одного запиту або продовжувати уже існуючий текст. Для роботи моделей використовуються великі обчислювальні потужності, а дані, використані для їхнього навчання, були зібрані з усього інтернету. OpenAI не публікує вагові параметри своїх моделей і не дає прямого доступу до них – підключення через API або інтегровані сервіси, наприклад, ChatGPT.

2.1.2. T5

Модель Text-to-Text Transfer Transformer, також відома як T5, була створена Google Research у рамках проекту "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". T5 використовується для обробки природної мови і використовує формат text-to-text, тобто вхідні і вихідні дані представлені у вигляді тексту, що спрощує роботу з моделлю [15]. На даний момент існує кілька конфігурацій моделі, зокрема Small, що містить близько 60M параметрів, Base (близько 220 мільйонів), Large (близько 770 мільйонів), 3B (3 мільярдів) і 11B (більше 11 мільярдів параметрів) [16].

Робочий процес T5 складається з двох етапів: попереднього навчання (pre-training) та тонкого налаштування (fine-tuning). Під час попереднього навчання модель навчається на величезному корпусі загальнодоступних текстових даних. Стандартна T5 навчена на корпусі C4, який складається з 745 ГБ фільтрованих текстів, очищених від нерелевантних даних, частин

коду та дублікатів. Метою навчання є передбачення замаскованих токенів у вхідному тексті, подібно до популярної моделі BERT. Однак, на відміну від BERT, де дані кодуються в числа, T5 використовує підхід тексту в текст, де і вхід, і вихід є текстовими представленнями [17].

Тонке налаштування тренує модель під конкретну задачу, надаючи завдання в текстовому вигляді, наприклад, «summarize: [text]». Наприклад, існує модифікація CodeT5, навчена на масивах коду та натренована для вирішення задач розробки програмного забезпечення.

Для виконання задач машинного навчання, особливо глибоких нейронних мереж, компанія Google розробила TPUs (Tensor Processing Units) – спеціальні прискорювачі, кожен з яких складається з великої кількості ядер, кожен з яких виконує обчислення в форматі тензорів і використовує принципи паралельності даних та паралельності моделі. TPUs використовуються у тому числі і у моделі T5. Окрім цього, використовується алгоритм Adafactor – оптимізатор, який доповнює TPU і дозволяє більш ефективно використовувати пам’ять, що особливо актуально при роботі з великими моделями та за умов браку ресурсів.

2.1.3. GNN

Графові нейромережі або GNN представляють дані у вигляді вузлів, ребер та додаткових атрибутів для них. GNN ефективні для задач, де важливі зв’язки між об’єктами: маршрутизації, мережах зв’язку, соціальних мережах, де користувача та його друзів можна представити як вузли у графі.

Вузли та ребра представлені у вигляді векторів. Для кожного вузла через ребра передається і об’єднується інформація із сусідніх за допомогою операцій min, max, sum, тощо. Отримавши агреговану інформацію, вузол оновлює власне представлення та інформацію про

зв’язки між вузлами у мережі. На кожному шарі мережі ці кроки повторюються.

Тим не менш, у графових мереж є ряд обмежень та недоліків. Вони потребують якісного парсеру даних, а розріджені графи можуть втрачати інформацію під час передачі.

GNN-моделі рідко застосовується для генерації опису коду. Проте їхнє використання дозволяє побудувати графи зв’язків між компонентами коду, зрозуміти, які модулі, функції чи класи залежать одне від одного [18].

Отже, з результатів роботи графової мережі можна сформувати графічну схему залежностей проекту і використати її як частину документації. До того ж, GNN можуть бути ефективними і у випадках використання у складних багаторівневих системах та можуть працювати на різних рівнях (функції, модулі, класи, файли).

2.2.Бібліотеки для роботи з ML-моделями

2.2.1. Hugging Face Transformers

Дана бібліотека є на сьогодні найпопулярнішою для роботи з трансформерами, вирішень задач їхнього налаштування та навчання. Сумісна з багатьма мовами програмування, зокрема Python, C++, TypeScript. Hugging Face Transformers підтримує моделі GPT, BERT, T5, RoBERTa, BART, тощо, а за потреби можна створити власну модель та навчати її з нуля [19]. Для токенізації вхідного тексту можна використати такі алгоритми як Byte Pair Encoding (BPE), WordPiece або SentencePiece, який використовувався при навчанні моделі T5.

Для навчання моделей існує клас Trainer, також доступний клас Seq2SeqTrainer для задач перетворення послідовностей. У клас Seq2SeqTrainer вбудовано механізми Length Penalty, щоб модель не генерувала занадто довгі або занадто короткі тексти та Beam Search для вибору найбільш ймовірної послідовності із можливих. Окрім цього,

додатково використовуючи бібліотеку datasets можна навчати модель на вже існуючих наборах даних або створити свій.

Очевидно, для навчання моделі у будь-якому випадку потрібні якісні дані, а сам процес може потребувати великих ресурсів, але загалом бібліотека зручна у користуванні і має велику спільноту користувачів та підтримку в середовищі розробників.

2.2.2. TensorFlow

Бібліотека була розроблена компанією Google для машинного навчання та обчислень. Основними одиницями даних в ній є тензори – багатовимірні блоки даних певного типу. Ранг тензора показує його вимірність, де 0 – скаляр, N – довільна кількість вимірів.

TensorFlow спочатку був побудований навколо концепції статичного графу обчислень, який можна було оптимізувати та запускати на різних пристроях. Вузлами графа були операції, частіше всього арифметичні, ребрами – тензори, які передаються між вузлами. Тепер у бібліотеці з’явилась можливість створити динамічну імперативну модель, яка не потребує побудови графа та дозволяє виконувати операції одразу.

Бібліотека також включає контекстний менеджер GradientTape, який відслідковує операції над тензорами та автоматично обчислює градієнти функцій втрат, що особливо важливо для навчання нейромереж [20]. Окрім цього, підтримується паралелізм даних та розподілення завдань між різними вузлами, що дозволяє раціонально розподілити навантаження. Також підтримується технологія TPUs та алгоритми оптимізації SGD і Adam, який є більш ранньою версією алгоритму Adafactor, згаданого раніше.

Створення моделей відбувається за допомогою високорівневого API keras. Модуль tf.data дозволяє завантажувати, опрацьовувати та керувати даними під час навчання моделі. Для паралельного виконання завдань

модуль використовує конвеєрну обробку (pipelines), а для швидшого виконання функцій мови Python їх можна перетворити у граф TensorFlow за допомогою модуля function. Модуль saved_model дозволяє зручно зберігати, імпортувати та експортувати моделі.

TensorFlow можна використовувати на мобільних пристроях та в мобільних веб-додатках через TensorFlow Lite і TensorFlow.js. Загалом бібліотека є обширною та гнучкою, проте може бути складною для розуміння.

2.2.3. PyTorch

Бібліотека PyTorch є менш популярною, ніж TensorFlow, проте останнім часом вона використовується все частіше. Вона розроблена компанією Meta, проста у використанні та дещо схожа з попередньою бібліотекою. Бібліотека представляє велику кількість уже попередньо навчених моделей для таких цілей як, наприклад, обробка природної мови чи комп'ютерний зір.

Як і в TensorFlow, основним блоком, з яким працює бібліотека, є об'єкт Tensor. Це багатовимірний масив, аналогічний тим, що використовуються в NumPy, але з додатковими можливостями роботи на графічних процесорах. Зокрема тензори підтримують складні функції матричної оптимізації, множення матриць та автоматичне диференціювання. Для останнього існує система Autograd, яка відслідковує всі операції, що відбуваються з тензорами, для побудови графа обчислень [21]. Для відслідковування і обчислення градієнтів кожен тензор обгортається класом torch.autograd.Variable.

Базовим класом для всіх нейронних мереж є nn.Module. Зазвичай моделі створюються наслідуванням від цього класу, а метод forward визначає, як дані переходять через різні шари нейромережі. Для оптимізації навчання можна використати алгоритми із пакету torch.optim,

які мінімізують функцію втрат: SGD, Adam, RMSProp, тощо. Модуль torch.utils.data.DataLoader дозволяє завантажувати дані пакетами, використовуючи при цьому багатозадачність, та трансформувати їх. Для роботи з зображеннями, текстами та звуковими даними існують підмодулі torchvision, torchtext і torchaudio відповідно. Як і у TensorFlow, підтримується розподілене навчання моделі на кількох вузлах через пакет torch.distributed, а для автоматичного розподілення навіть існує система torch.nn.DataParallel.

Загалом бібліотека PyTorch є досить гнучкою та дозволяє працювати з широким вибором моделей: від рекурентних до трансформерів. Для різних, навіть вузькоспеціалізованих задач, існують окремі бібліотеки та пакети, які полегшують роботу розробника. З мінусів бібліотеки можна виділити хіба що меншу підтримку та рідкі випадки використання на мобільних пристроях.

2.3.Бібліотеки для аналізу коду

2.3.1. ANTLR

Дана бібліотека використовується для створення інтерпретаторів, парсерів, компіляторів, тощо. На вхід приймається граматика у форматі BNF (Backus-Naur Form) і на виході формуються лексичний та синтаксичний аналізатори.

ANTLR підтримує велику кількість різних мов програмування: Go, C, Swift, Python та інші [22]. Зчитану структуру коду можна представити у вигляді синтаксичних дерев [23], для роботи з ними підтримуються патерни Listener і Visitor. Останній підхід є більш гнучким, адже розробник сам керує обходом дерева, а патерн корисний для перетворення дерев або генерації вихідного коду.

У ANTLR підтримується стратегія LL(*), у якій число символів попереду, які треба переглянути для точного визначення синтаксичної

структури, є змінним, на відміну від LL(k), де кількість символів є фіксованою. Це вагома перевага ANTLR над іншими бібліотеками, адже деякі мови програмування мають складні граматики і стандартний механізм LL(k) менш ефективний у роботі з ними.

Загалом бібліотека має широку сферу застосування. Вона ефективна не лише для аналізу та рефакторингу коду, написаного на популярних мовах програмування, а й для парсингу файлів типу XML та JSON і файлів конфігурації. Можлива навіть розробка спеціалізованих мов для конфігурації або аналізу даних.

2.3.2. Libclang

Libclang надає доступ до багатофункціонального компілятора Clang через API і відповідно підтримує мови C, C++, Objective-C і Objective-C++. Задачі бібліотеки включають у себе синтаксичний та лексичний аналіз, побудову AST, пошук по коду, генерацію діагностичних повідомлень та попереджень, тощо [24].

Основним елементом бібліотеки є Translation Unit (TU), тобто файл вихідного коду з його залежностями. Після аналізу TU будується абстрактне синтаксичне дерево, яке відображає синтаксичну структуру коду. Вузли дерева представляють елементи коду – функції, класи, змінні. У кожному дереві існує курсор, який переміщається між вузлами і дозволяє отримувати інформацію про елементи коду та аналізувати залежності між ними. Також Libclang має вбудований механізм для обробки помилок під час аналізу коду. Окрім цього, бібліотека може витягувати з коду структуру та коментарі для подальшого створення документації.

Деякі середовища розробки та редактори коду, такі як Visual Studio та CLion, використовують Libclang для підсвічування синтаксису та статичного аналізу коду. Незважаючи на простоту використання навіть у великих проектах, високу продуктивність та підтримку багатьох платформ,

бібліотека має мінуси – вона надає доступ до неповного списку можливостей Clang та залежна від файлової системи, але тим не менш залишається потужним інструментом для роботи з C-подібними мовами.

2.3.3. Esprima

Esprima є невеликою бібліотекою, написаною для мови JavaScript, основною задачею якої є побудова AST. Вона вузькоспеціалізована, створена лише для Javascript і не може використовуватись для інших мов, проте вона підтримує всі специфікації мови, навіть найновіші ECMAScript 6+ або ESNext.

Синтаксичні дерева, які будує Esprima, відповідають специфікації ESTree, яка є стандартом для роботи з JavaScript [25]. Через це вони активно використовуються у бібліотеці ESLint для аналізу коду на відповідність стандартам.

Espima добре інтегрується з Babel та Terser і часто використовується для розширень, які аналізують або модифікають код. Бібліотека може аналізувати не лише цілий файл, а й окремий фрагмент, а також підтримує строгий режим, імпорт та експорт модулів, асинхронність, представлення результатів аналізу в форматі JSON, тощо. До списку можливостей бібліотеки входить пошук синтаксичних помилок з поверненням повної інформації про її причину та місцезнаходження.

Отже, даний інструмент є вузьконаправленим і менш гнучким, ніж, наприклад, Babel, але може бути ефективним для рефакторингу коду або створення документації.

2.4. Бібліотеки для генерації графів залежностей

2.4.1. NetworkX

NetworkX – бібліотека для мови Python, створена для проектування, зміни та вивчення мереж, представлених у вигляді графів. Підтримує орієнтовані, неорієнтовані та мульти-графи. Бібліотека має ряд алгоритмів для пошуку інформації в графі, а також його аналізу. Зокрема підтримується алгоритм Беллмана-Форда, пошук в глибину та ширину, оцінка центрів графа, тощо [26]. Для всіх вузлів та ребер можна створювати додаткові атрибути.

NetworkX може бути корисною для візуалізації графів, особливо з використанням додаткових бібліотек. Модулю Matplotlib може бути достатньо для простих графів, тоді як Plotly дозволяє створювати більш складні та інтерактивні схеми. Тому бібліотека NetworkX у зв’язці з GNN може бути використана для візуалізації залежностей між частинами проекту у створеній документації.

2.4.2. Graph-tool

Бібліотека Graph-tool є більш продуктивним аналогом NetworkX та використовується у випадку роботи з великими графами, де нараховуються мільйони вузлів.

Graph-tool в основному частково написана на Python, але більшість ключових операцій реалізовано на C++ з використанням бібліотеки Boost Graph Library для підвищення продуктивності та швидкості [26, 27]. Загалом Graph-tool підтримує всі ті ж самі алгоритми для обходу та аналізу графів, що і попередня бібліотека, та підтримує паралелізм. Окрім цього, Graph-tool можна використовувати для підготовки даних для машинного навчання, зокрема для генерації матриць суміжності графів, обчислення

показників графів (центральності, метрик кластеризації) та аналізу тимчасових графів.

Проте значною перевагою інструменту є наявність вбудованих можливостей візуалізації графів. Graph-tool надає API для додавання властивостей (атрибутів) до вершин, ребер і графу в цілому. Розробник може змінювати позиції вузлів, товщину ребер та їхній колір.

Підтримується експорт графів у форматах GraphML, PDF, PNG, SVG Dot, що дозволяє легко вставляти їх у необхідні документи.

2.5. Інструменти для створення вихідного документа

2.5.1. Jinja2

Шаблонізатор Jinja2 призначений для створення файлів HTML, XML, JSON і тому подібних у програмах на мові Python. Jinja2 дозволяє змішувати статичний текст з динамічним вмістом, що передається через змінні, котрі вставляються в текст через просту нотацію [28].

Спочатку Jinja2 обробляє шаблон, перетворюючи його на AST, далі це дерево представляє статичний та динамічний текст у вигляді вузлів. Створене AST компілюється у Python-код. Так шаблон по суті стає виконуваним Python-кодом і це пришвидшує подальший рендеринг. У кінці скомпільований шаблон інтерпретує передані дані та генерує вихідний текст.

Шаблонізатор підтримує умовні конструкції та цикли, підтримує форматування даних перед виведенням і забезпечує гнучкий та інтуїтивний синтаксис, аналогічний Django Template Language. Jinja2 підтримує модульний підхід, що дозволяє використовувати блоки повторно і робити код більш читабельним.

Інструмент вбудований у фреймворк Flask у якості шаблонізатора, а також добре інтегрується з FastAPI та Django, він є гнучким та має високу

продуктивність. Проте для великих файлів або тих, які потребують вагомих обчислень, може знадобитись оптимізація.

2.5.2. Mako

Шаблонізатор Mako призначений для створення динамічних текстових файлів та є кращим для великих та складних файлів, ніж Jinja2. Шаблони Mako підтримують пряме виконання коду Python, а для прискорення обробки вони компілюються в байт-код Python. Також шаблонізатор має вбудовані механізми для захисту від XSS-атак.

Всі шаблони зберігаються у файлах з розширенням .mako, дані передаються до них через словники [29]. Змінні вставляються у шаблони через просту нотацію, також є можливість вставки фрагментів коду та визначення власних функцій у шаблонах. Розробник може за потреби створити власні обробники даних. Як і у попередньому шаблонізаторі, у Mako підтримується модульний підхід та наслідування, також можна імпортувати один файл шаблону в інший.

Хоча шаблонізатор Mako є менш популярним, ніж Jinja2, його перевагами є швидкість, підтримка кешування, масштабованість та простота у використанні.

2.6. Запропонований підхід

Створення генератора опису коду – це комплексне завдання, одна з основних задач якого – вибір правильних бібліотек та технологій для успішного функціонування системи. Проаналізувавши усі існуючі інструменти та можливості їхнього використання було обрано наступний набір технологій:

1) Попередньо навчені моделі T5 та GNN для написання анотацій для коду та аналізу залежностей між компонентами коду відповідно;
2) Інтеграція з GitHub REST API для отримання публічних репозиторіїв та змін в них;
3) ANTLR для парсингу коду, побудови AST і формування токенів для донавчання T5 з метою тонкого налаштування моделі для виконання задач, пов’язаних зі створенням анотацій;
4) PyTorch для самого процесу навчання моделей;
5) Бібліотека Celery для налаштування асинхронного виконання коду з метою підвищення його швидкодії;
6) Graph-tool для візуалізації графів залежностей, сформованих GNN;
7) Mako для формування документу з документацією у форматі HTML;
8) Веб-фреймворки ReactJS та Flask для організації фронтенду та бекенду додатка.

Отже, результатом виконання даної роботи буде система, основною задачею якої буде формування документації до репозиторію GitHub, посилання на який надає користувач. Оскільки документація формується за допомогою моделей машинного навчання, вона зможе формувати пояснення коду більш точно у порівнянні з інструментами, котрі формують пояснення з шаблонів. Створення та візуалізація графів залежностей дозволить наглядно продемонструвати користувачу структуру проекту для кращого його розуміння, а HTML-формат файлів документації зручний для перегляду та взаємодії. Використання фреймворків ReactJS та Flask дозволить створити односторінковий сайт, який буде приймати посилання на репозиторій та повертати користувачу сформовану документацію.

ВИСНОВКИ ДО РОЗДІЛУ 2

У даному розділі було розглянуто існуючі інструменти для роботи з кодом, моделями машинного навчання та графами. На основі вивченої інформації було запропоновано власний підхід з використанням сучасних технологій.

Модель T5 працює у форматі «text-to-text», що є ефективним форматом для обраної задачі, та навчена на великому обсягу даних. Відповіді моделі T5 більш строгі та структуровані, аніж GPT, котра схильна до більш вільних генерацій. До того ж, T5 рідше використовується в уже існуючих рішеннях, а тому її застосування може мати певний потенціал. Моделі типу GNN найкраще підходять для аналізу, створення та модифікації графів і будуть ефективним рішенням для візуалізації роботи програми.

Бібліотека PyTorch має широкий набір інструментів для навчання мереж і широко використовується для роботи з ними, а модуль PyTorch Geometric підтримує навіть роботу з графами. Інструмент ANTLR підтримує велику кількість мов програмування і використовує стратегія LL(*), тому є більш ефективним, ніж аналоги. Шаблонізатор Mako має ряд переваг, серед яких висока продуктивність, передбачений захист від XSS-загроз та хороша інтеграція з іншими інструментами.

3. ТЕОРЕТИЧНІ ВІДОМОСТІ ЩОДО ОСНОВНИХ СКЛАДОВИХ СИСТЕМИ

3.1. Особливості структури та навчання моделі T5

3.1.1. Загальні відомості про модель T5

Модель T5 (Text-to-Text Transfer Transformer) була запропонована компанією Google Research у 2020 році в роботі “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”. Її ключова ідея полягає у тому, що всі завдання обробки природної мови (NLP) – переклад, класифікація, резюмування, заповнення пропусків тощо – можна звести до єдиного формату “текст → текст” [30].

Таким чином, T5 приймає на вхід текстову послідовність (наприклад, уривок коду або опис задачі) і генерує текстову відповідь (наприклад, коментар до коду або пояснення логіки функції).

Архітектурно T5 базується на класичному трансформері, який складається з двох основних частин:

1) Енкодера, що зчитує вхідну послідовність і перетворює її у внутрішнє векторне представлення;
2) Декодера, який на основі цього представлення генерує вихідний текст.

Обидві частини містять багатошарові блоки з механізмом самоуваги (self-attention), що дозволяє моделі враховувати контекст усіх токенів послідовності одночасно.

Використана для даного дослідження версія T5-base має ~220 мільйонів параметрів, 12 шарів кодера і 12 шарів декодера, що забезпечує достатню глибину для захоплення складних залежностей в тексті і коді. Розмір прихованого стану становить 768, тобто кожна позиція маркера закодована у векторі фіксованої розмірності 768. Модель має 12 голів уваги, що дозволяє моделі ефективно розподіляти увагу між різними токенами у вхідній

послідовності. Довжина ембедингів токенів 768, розмір шару прямого розповсюдження, що застосовується всередині кожного шару трансформатора для обробки контекстної інформації, становить 3072.

3.1.2. Механізм уваги та функція втрат

Однією з ключових інновацій архітектури трансформера, на якій побудована модель T5, є механізм уваги. Його головна ідея полягає в тому, щоб модель могла динамічно визначати, які частини вхідної послідовності є найбільш важливими для поточного прогнозу.

Традиційні рекурентні нейронні мережі (RNN, LSTM) обробляють текст послідовно – токен за токеном, тому вони мають труднощі з довгими залежностями у тексті.

Механізм уваги вирішує цю проблему: він дозволяє моделі паралельно аналізувати всі токени і визначати, які з них несуть найбільше інформації для поточного слова чи фрагмента коду.

Кожен токен у вхідній послідовності представлений трьома векторами:

1) Query (Q) – запит, який описує, що саме шукає даний токен у контексті;
2) Key (K) – ключ, що описує, яку інформацію цей токен може надати іншим;
3) Value (V) – значення, тобто саме інформаційне представлення токена.

Механізм уваги обчислює схожість між кожним запитом (Q) і всіма ключами (K), визначаючи, наскільки важливий кожен токен контексту для поточного. У виді формули це виглядає так:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V, \quad (3.1)$$

де QK^T – міра подібності між запитами і ключами, d_k – розмірність векторів ключів.

Функція softmax перетворює значення подібності у ймовірності, які показують «ступінь уваги» до кожного токена, а множення на V дозволяє об'єднати цю інформацію у зважене середнє – підсумкове представлення контексту для даного токена.

Щоб навчитися одночасно враховувати різні типи зв'язків між токенами зокрема лексичні, синтаксичні і семантичні, трансформер використовує багатоголову увагу.

Замість одного набору векторів Q, K, V модель створює кілька таких наборів («голів» уваги), кожна з яких навчається фокусуватися на певному аспекті контексту (3.2).

$$MultiHead(Q, K, V) = Concat(head_1, head_2, \dots, head_h)W^O, \quad (3.2)$$

де

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V, \quad (3.3)$$

а W_i^Q, W_i^K, W_i^V – це навчальні матриці перетворення для кожної голови.

Завдяки цьому підходу модель T5 здатна ефективно розуміти як локальні залежності, між словами в межах одного виразу, так і глобальні, наприклад, між різними частинами коду чи речення.

У моделі T5 використовуються два різновиди уваги:

1) Self-Attention (самоувага) – застосовується в енкодері й декодері окремо. Кожен токен «уважно» аналізує інші токени тієї ж послідовності, щоб зрозуміти контекст.
2) Cross-Attention (перехресна увага) – використовується у декодері для зв'язку з енкодером. Таким чином, декодер має доступ до представлення всього вхідного тексту, згенерованого енкодером.

Це дає змогу моделі в T5 генерувати вихідний текст, спираючись на повний контекст вхідного коду, що є особливо важливим для завдань пояснення або коментування коду.

Під час обробки програмного коду кожен токен (наприклад, ідентифікатор функції, ключове слово return, чи символ if) «звертає увагу» на інші частини коду. Наприклад, при аналізі return result модель може приділити найбільше уваги тому, де саме оголошено змінну result, щоб правильно зрозуміти її контекст і описати у коментарі її призначення.

Наступним ключовим елементом процесу навчання моделей обробки природної мови є функція втрат, що вимірює відхилення між передбаченнями моделі та правильними еталонними значеннями.

Для моделей типу T5, які генерують послідовності тексту, традиційно використовується функція крос-ентропійних втрат. T5 навчається на основі підходу sequence-to-sequence, тобто для кожної вхідної послідовності X вона повинна відтворити вихідну послідовність Y.

Формально модель намагається визначити різницю між передбаченим розподілом ймовірностей та правильним мітковим розподілом (3.4):

$$L = -\sum_{i=1}^{N} y_i \log(p_i), \tag{3.4}$$

де y_i – справжнє значення (1 для правильного токена, 0 для решти), а p_i – передбачена ймовірність токена.

Ця функція вимагає посиленої уваги за відхиленнями від правильної послідовності: навіть незначна помилка у токені суттєво впливає на загальний результат.

Втрата фактично вимірює «відстань» між розподілом, передбаченим моделлю, і реальним розподілом, що відповідає правильним токенам. Мінімізація цієї втрати означає, що модель стає все точнішою у прогнозуванні правильних токенів, тобто здатна формувати більш осмислені, зв’язні та релевантні тексти.

3.1.3. Токенізація за допомогою SentencePiece

Перш ніж текст або код можна буде подати у модель, його необхідно перетворити на числову форму, з якою може працювати нейронна мережа. Цей процес називається токенізацією, а інструмент, який її виконує, – токенізатор.

У моделі T5 токенізація виконується за допомогою бібліотеки SentencePiece, розробленої в Google. SentencePiece – це мовонезалежний токенізатор, який може працювати з будь-яким текстом, включно з кодом, де немає чітких роздільників між «словами».

SentencePiece спочатку розглядає текст як неперервну послідовність символів Unicode (без поділу на слова). Потім він навчає статистичну модель для визначення найбільш імовірних підпослідовностей (саб токенів), які найчастіше зустрічаються у корпусі. Зазвичай це робиться з використанням одного з двох алгоритмів:

1) Byte-Pair Encoding (BPE) – поступово об’єднує найчастіше зустрічані пари символів або токенів;
2) Unigram Language Model – знаходить оптимальний набір підпослідовностей, що мінімізують загальну ентропію корпусу.

Результатом є словник токенів, де можуть бути як цілі слова, так і їхні частини, наприклад: "import" → ["im", "port"]; "variable" → ["vari", "able"].

Серед переваг SentencePiece можна зазначити, що він не потребує попередньої сегментації тексту, а тому підходить для багатомовних систем або для програмного коду, де відсутні пробіли та можуть бути специфічні символи. Завдяки компресії словника зменшується кількість рідкісних токенів, що спрощує навчання моделі.

У випадку із програмним кодом SentencePiece дозволяє зберігати структуру, не розриваючи важливі частини, як ті ж ідентифікатори (calculate_sum → calculate, _, sum).

Токенізатор підтримує додаткові службові токени, наприклад:

1) <pad> – для заповнення порожніх позицій;

2) <eos> – для позначення кінця послідовності;

3) <unk> – для невідомих токенів.

У проєкті токенізатор SentencePiece застосовується для попередньої обробки вихідного коду перед подачею в модель T5. Кожен фрагмент коду очищується від коментарів та непотрібних пробілів, перетворюється у послідовність токенів SentencePiece, а далі переводиться у числові індекси, які потім подаються до вхідного шару T5.

Зворотний процес – це декодування, під час якого числові послідовності перетворюються назад у текст, тобто у зрозумілі людині коментарі до коду.

3.1.4. Оптимізатор AdamW

Adam комбінує ідеї двох популярних методів:

1) Momentum, який враховує накопичену історію градієнтів, щоб рух у просторі параметрів був більш стабільним;
2) RMSProp, який масштабує крок навчання окремо для кожного параметра залежно від дисперсії його градієнта.

Формули оновлення параметрів у Adam:

$$w_{t+1} = w_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} - \eta \cdot \lambda w_t, \tag{3.5}$$

де $\hat{m}_t$, $\hat{v}_t$ – експоненціальні середні градієнтів і квадратів градієнтів, η – швидкість навчання, λ – коефіцієнт зниження ваги.

Основною особливістю AdamW є введення зменшення ваг як окремого додаткового кроку, а не як частину градієнта. Це усуває теоретичну проблему класичного Adam, де регуляризація впливала на напрям градієнта, що могло викликати переобладнання.

Перевагами AdamW є стабільність та швидка збіжність навіть на великих наборах даних, ефективність при тренуванні трансформерів, а також покращення узагальнення та контроль над переобладнанням.

3.1.5. Застосування T5 у проєкті

Для донавчання моделі було використано метод LoRA, який дозволяє адаптувати великі трансформери, вводячи додаткові матриці низького рангу до ваг ключових шарів моделі. Зазвичай це роблять для ваг query та value у механізмі attention. Особливостями методу є те, що основні параметри моделі заморожуються, натомість додаються менші адаптерні матриці A та B (3.6). Вихід кожного шару змінюється таким чином:

$$W' = W + \Delta W = W + A \cdot B \,, \tag{3.6}$$

де W – незмінні ваги оригінальної моделі, ΔW – навчувані матриці низького рангу.

Градієнти обчислюються лише для цих матриць, а решта моделі лишається незмінною (рисунок 3.1). Це дозволяє моделі швидко підлаштовуватися під нову доменну інформацію при дуже малому додатковому обсязі пам’яті.

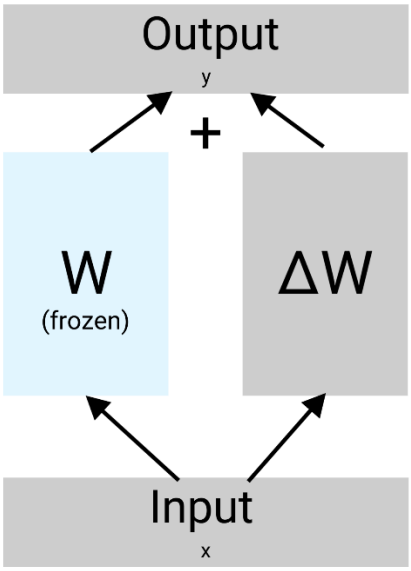

Рисунок 3.1 – Формування вихідного значення за допомогою алгоритму LoRA

Такий метод може трохи поступатися у точності повному fine-tuning і не завжди підходить для завдань, де потрібна глибока модифікація всіх шарів моделі. Проте LoRA компенсує свої недоліки тим, що у рази зменшує потребу в пам’яті у процесі навчання та забезпечує швидке тренування, оскільки оновлюються лише додаткові параметри. Такий метод підходить навіть для великих моделей.

Метод LoRA часто застосовують для адаптації моделі на невеликому наборі даних та загалом для донавчання для вузьконаправленої задачі. Наприклад, для створення спеціалізованих чат-ботів або донавчання під специфічний стиль тексту.

У даному випадку було доцільним використати саме цей підхід, адже наявні ресурси були обмеженими і створення моделі з нуля або навіть повний fine-tuning були б складними і не дали бажаного результату. Модель T5-base вже навчена на великому корпусі очищених даних і еефективно працює з текстом, тому часткова її адаптація для задач коментування коду можна вважати раціональним з точки зору використання ресурсів. Завдяки замороженню основних параметрів T5 зберігається знання про загальні мовні

закономірності та семантику коду, що дозволяє моделі генерувати коментарі більш природно та інформативно.

У процесі інтеграції методу LoRA для донавчання моделі T5 ключовим елементом є правильне налаштування параметрів адаптера, що визначають його поведінку та ефективність. Конфігурація LoRA задається об’єктом LoraConfig, який містить кілька важливих параметрів.

Параметр task_type=TaskType.SEQ_2_SEQ_LM вказує на тип завдання, для якого здійснюється донавчання. У даному випадку це послідовність-до-послідовності (sequence-to-sequence), тобто модель навчається генерувати вихідну послідовність токенів на основі вхідної послідовності токенів коду. Цей параметр визначає, як LoRA буде інтегруватися у шари моделі та обробляти інформацію в контексті генеративного завдання.

Параметр r=8 визначає ранг адаптера, тобто розмір додаткових матриць низького рангу, які додаються до основних ваг attention. Менший ранг забезпечує економію пам’яті та швидкість тренування, але може обмежувати здатність моделі адаптуватися до складних особливостей нових даних. Значення r=8 є компромісом між ефективністю адаптації та ресурсозбереженням.

Параметр lora_alpha=32 визначає шкалу адаптації, тобто множник, який контролює вплив додаткових низькорангових матриць на вихід основного шару. Більше значення alpha збільшує вплив LoRA на модель, дозволяючи швидше адаптуватися до специфічних особливостей завдання, тоді як менше значення робить адаптацію більш консервативною.

Параметр target_modules=["q", "v"] вказує, у які шари механізму multi-head attention будуть інтегровані адаптерні матриці. У трансформерах саме шари query та value відіграють ключову роль у формуванні контекстної уваги, тому адаптація саме цих компонентів забезпечує ефективне впливання на репрезентації без зміни всієї моделі.

Нарешті, lora_dropout=0.1 задає ймовірність випадкового відключення елементів адаптера під час навчання, що є аналогом стандартного dropout для

нейронних мереж. Цей механізм дозволяє уникнути перенавчання, особливо коли навчальні дані мають обмежений обсяг або недостатньо репрезентативні.

Таким чином, дана конфігурація LoRA забезпечує оптимальне поєднання адаптивності, стабільності та економії ресурсів, дозволяючи ефективно донавчати T5 на завданні генерації коментарів до коду без необхідності змінювати всі параметри великої трансформерної моделі.

Для токенізації використовується регулярний SentencePiece-токенізатор моделі T5 – T5TokenizerFast, який перетворює вхідні тексти в послідовності токенів та їх числові індекси. SentencePiece побудований на принципах підсимвольного моделювання та працює без попередньої сегментації тексту, що робить його універсальним для будь-яких вхідних мов, включно з фрагментами коду.

Вхідні дані формуються у вигляді структурованої текстової інструкції типу "summarize code: <код> AST: <лінійне представлення AST> GRAPH: <опис графових ознак>"

Таке структурування з використанням ключових маркерів допомагає моделі розрізняти різні частини вхідних даних та асоціювати їх із відповідними типами інформації. Коментар токенізується окремо. Токени заповнення замінюються на спеціальний індекс -100, щоби не враховувати їх під час обчислення функції втрат.

Обмеження m,ax_input_length встановлено на рівні 512 токенів, а max_target_length – 64 токени. У випадку, коли фрагмент коду занадто довгий, застосовується метод обрізання.

3.2. Особливості структури та навчання моделей типу GNN

3.2.1. Загальні відомості про графові нейронні мережі

Графові нейронні мережі є класом моделей, розроблених для обробки даних у вигляді графів. На відміну від традиційних нейронних мереж, які

працюють з регулярними структурами, такими як послідовності або сітки, GNN оперують складними нерегулярними структурами, де об’єкти представлені у вигляді вузлів, а зв’язки між ними – у вигляді ребер. Це дозволяє моделю ефективно захоплювати залежності між елементами даних, які не можна подати у вигляді послідовності або матриці. У контексті аналізу коду та побудови схем залежностей між файлами або модулями GNN особливо корисні, оскільки структура коду природно формалізується як граф: вузли – це функції, класи або модулі, а ребра – імпорти, виклики функцій або посилання на змінні [31].

Основна ідея GNN полягає в агрегації інформації від сусідніх вузлів. На кожній ітерації, тобто у кожному шарі, мережа оновлює представлення кожного вузла, враховуючи ознаки його сусідів. Цей процес називається «message passing» і має таку формулу (3.7):

$$h_v^{(k)} = \text{UPDATE}^{(k)}(h_v^{(k-1)}, \text{AGGREGATE}^{(k)}(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\})), \quad (3.7)$$

де $h_v^{(k)}$ – векторне представлення вузла v на k-му шарі, $\mathcal{N}(v)$ – множина сусідів вузла, функції AGGREGATE та UPDATE – навчені оператори (наприклад, середнє), які збирають інформацію та оновлюють стан вузла.

Після кількох шарів агрегації кожен вузол отримує контекстне представлення, яке враховує структуру графа навколо нього.

Архітектура GNN зазвичай будується на основі кількох ключових компонентів. По-перше, представлення вузлів – це вектори, які описують властивості кожного вузла. Для коду це можуть бути вектори, що кодують ім’я функції, тип змінної або токенізовану інформацію з AST. По-друге, агрегаційні функції, які дозволяють об’єднувати інформацію від сусідів кожного вузла для формування нового представлення вузла. Агрегація може здійснюватися різними способами: сумуванням, усередненням, максимізацією або більш складними механізмами на кшталт attention. Це дозволяє вузлу

отримувати контекст з оточення і враховувати взаємозв’язки в графі. По-третє, оновлювальні функції, які комбінують власне представлення вузла з агрегованою інформацією від сусідів для формування нового embedding. Такі оновлення повторюються кілька разів, створюючи багаторівневе представлення вузлів з урахуванням ширшого контексту графа.

Існує кілька класичних архітектур GNN, які використовуються для різних задач. Graph Convolutional Networks (GCN) застосовують принцип згортки до графів, де нове представлення вузла обчислюється як лінійна комбінація представлень сусідніх вузлів із подальшою нелінійною активацією. GCN добре підходять для задач класифікації вузлів або предикції зв’язків. Graph Attention Networks (GAT) використовують механізм уваги для обчислення ваг сусідів під час агрегації, що дозволяє моделі зосереджуватися на більш важливих вузлах та ігнорувати неінформативні. Це особливо корисно у великих графах коду, де не всі імпорти чи виклики функцій однаково значущі для конкретного завдання, саме тому ця архітектура була обрана для даного проекту. GraphSAGE – ще одна популярна архітектура, яка навчає функції агрегації для вузлів на локальних підграфах і дозволяє ефективно масштабуватися на великі графи.

Особливості GNN у контексті аналізу коду мають декілька важливих моментів. По-перше, подання графа коду: вузли можуть представляти функції, класи, модулі або навіть окремі змінні, а ребра – виклики функцій, імпорти, залежності між змінними або успадкування. Така формалізація дозволяє моделі ефективно виявляти структури та взаємозв’язки, які неявно присутні у тексті коду.

По-друге, обробка великих графів: у проектах з багатьма файлами та класами графи можуть бути дуже великими, що вимагає використання підграфів або sampling-технік для агрегації інформації, наприклад, Neighbor Sampling у GraphSAGE. По-третє, фічі вузлів і ребер: для підвищення точності моделі можна додавати атрибути, такі як типи даних, кількість параметрів функції, кількість викликів або метрики складності коду.

Навчання GNN зазвичай відбувається за допомогою зворотнього поширення по графу, використовуючи стандартні функції втрат, такі як Cross-Entropy Loss для задач класифікації вузлів або MSELoss для регресії. Важливою частиною процесу є нормалізація та регуляризація, щоб уникнути проблем із затуханням або вибухом градієнтів, оскільки багато графів у коді мають вузли з великою кількістю сусідів. Для цього часто застосовують нормалізацію Batch Normalization, яка стабілізує розподіл представлень вузлів у батчах, і Dropout для ребер або вузлів для уникнення перенавчання.

У даному проекті GNN застосовується для генерації графів імпортів і експортів, де вузли – окремі модулі або файли, а ребра – імпортовані або експортовані об’єкти. Архітектура мережі складається з 4 шарів GAT, кожен із яких забезпечує поширення інформації на k-hop сусідів, де k визначає глибину аналізу залежностей. Вихідні ембедінги вузлів використані для візуалізації графа за допомогою бібліотеки Graph-tool. Таким чином, модель здатна виділяти ключові модулі та виявляти залежності, що оптимізує подальший процес автоматичного документування або аналізу архітектури програмного забезпечення.

Переваги використання GNN у цьому контексті включають можливість моделювати складні залежності коду, масштабуватися на великі проєкти, інтегруватися з різними джерелами даних (AST, ембедінги файлів, метадані) і забезпечувати високу точність при побудові графів залежностей. Недоліки полягають у складності налаштування архітектури та високих обчислювальних витратах при роботі з великими графами.

3.2.2. Особливості структури створеної GNN

У рамках цієї роботи була розроблена та навчена графова нейронна мережа (GNN) для аналізу залежностей між файлами та кодовими модулями з метою побудови діаграм імпорту та експорту.

Представлення коду формується таким чином: кожен файл або модуль розглядається як вузол графа, а ребра між вузлами відповідають залежностям, таким як імпорт, виклики функцій або посилання на класи. Кожне з'єднання може бути додатково закодовано з характеристиками, що відображають його тип і важливість. Кожен вузол має вкладення, які відображають його характеристики: наявність docstring, кількість функцій і класів, довжину коду і частоту виклику функцій. Ребра кодуються за допомогою двійкових індикаторів типів залежностей, що дає змогу розрізняти ключові імпорти та вторинні посилання.

Для архітектури моделі було обрано багатошарову Graph Attention Network (GAT), що складається з чотирьох шарів, кожен з яких агрегує інформацію від сусідніх вузлів за допомогою механізму уваги. Такий підхід дозволяє диференціювати важливість різних залежностей, забезпечуючи більш точне представлення структури проекту. оскільки він дозволяє призначати різні ваги сусіднім вузлам при агрегуванні інформації. Це особливо важливо в задачах аналізу коду, де деякі імпорти або ключові функції мають набагато більший вплив на структуру та поведінку програми. Послідовні шарів GAT виконують агрегацію ознак сусідніх вузлів з подальшим застосуванням нелінійної функції активації ReLU. Між шарами використовується нормалізація BatchNorm і регуляризація, що забезпечує стабільність навчання і запобігає перенавчанню. На вихідному рівні модель генерує вкладення вузлів, які потім використовуються для побудови візуальної діаграми залежностей.

Навчання моделі проводилося за допомогою оптимізатора AdamW з швидкістю навчання 1e-3 і зниженням ваги 1e-5. Cross-Entropy Loss використовувався як функція втрат для класифікації вузлів за важливістю та ідентифікації ключових залежностей. Щоб запобігти перетренованості, на вузли та ребра наносили відсівання 0, 1. Для великих графів застосовувалося пакетування з torch_geometri at ataLoader, що дозволило ефективно тренувати модель на декількох файлах і сховищах одночасно.

Під час навчання були обрані оптимальні параметри: чотири шари ГАТ, 8 голівок уваги в кожному шарі, розмір вбудовувань вузлів 128 і відсівання 0, 1. Ця конфігурація дозволила моделям точно визначити ключові залежності, включаючи важливі імпорт і функції, а також сформувати схему проекту в графік, придатний для візуалізації та подальшої генерації документації. В результаті навчений GNN забезпечує високу точність передбачень залежностей і є ефективним інструментом для побудови структурованих схем коду, інтегрованих в систему автоматичної генерації документів, що використовується в цій роботі.

3.3. Побудова синтаксичного дерева

3.3.1. Роль генератора ANTLR у проекті

ANTLR – це генератор парсерів, який суттєво спрощує роботу з мовами програмування, формальними граматиками та побудовою синтаксичних дерев. У контексті проєкту, де потрібно отримати структуроване уявлення Python-коду для подальшого аналізу, формування абстрактного синтаксичного дерева (AST) та генерації семантичних описів, ANTLR виступає ключовим інструментом, що забезпечує формальну, універсальну та машиночитану інтерпретацію вихідного коду.

ANTLR грунтується на ідеї перетворення формального опису граматики мови на повноцінний парсер. Граматика визначає, з яких елементів складається мова (лексеми, правила, синтаксичні структури), і ANTLR автоматично генерує код, який виконує лексичний та синтаксичний аналіз. У традиційному підході розробники вручну створюють лексери, парсери, обробку помилок, ітеративно тестують граматику, підлаштовують структуру дерева. ANTLR позбавляє від цих складностей, забезпечуючи один із найбільш надійних та перевірених механізмів побудови parse tree.

ANTLR працює у два етапи: лексичний аналіз (лексер) та синтаксичний аналіз (парсер). Лексер відповідає за перетворення сирцевого тексту на токени – мінімальні одиниці синтаксису: ключові слова Python, ідентифікатори, літерали, дужки, оператори.

Парсер, використовуючи послідовність токенів, будує ієрархічне дерево відповідно до правил граматики. У результаті формується parse tree – структуроване представлення коду, де кожен вузол відповідає певній синтаксичній конструкції: визначення функції, імпорт, оголошення класу, блок умовного оператора, цикл (рисунок 3.2).

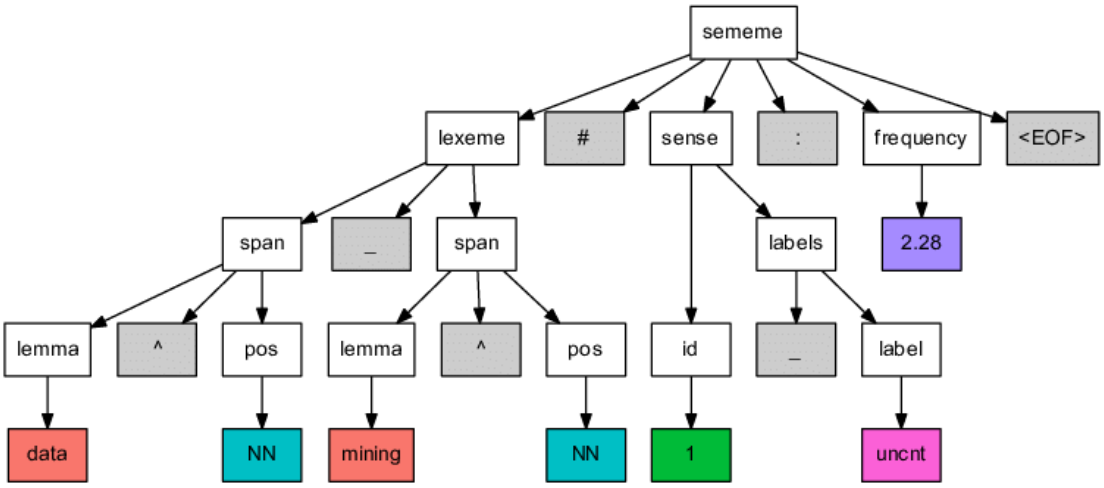

Рисунок 3.2 – Приклад отриманого AST

Важливо підкреслити, що ANTLR використовує LL(*)-парсинг – узагальнення LL-парсерів, здатне передбачати кілька токенів наперед і коригувати рішення залежно від контексту. Це дає можливість ефективно парсити навіть складні циклічні та рекурсивні структури Python. У багатьох мовах програмування виникають неоднозначні конструкції (наприклад, у Python – залежність структури блоків від відступів), і ANTLR ефективно розв’язує такі колізії завдяки власній оптимізованій моделі обробки граматик.

Попри численні переваги ANTLR, його застосування для аналізу Python-коду має низку особливостей та потенційних недоліків, які важливо враховувати під час розробки системи статичного аналізу та побудови AST. Python – мова з унікальним синтаксисом, який ускладнює використання

традиційних LL-парсерів, і хоча існуюча граматика для ANTLR здебільшого розв’язує ці труднощі, певні аспекти можуть вплинути на якість, масштабованість чи швидкодію аналізу.

На відміну від більшості мов, де структура блоків визначається фігурними дужками, Python використовує відступи як синтаксичну частину мови. Це ускладнює побудову граматик, оскільки парсер має не просто розпізнавати токени, а фактично відтворювати логіку Python-інтерпретатора, який перетворює відступи на спеціальні токени INDENT та DEDENT .

ANTLR підтримує подібну функціональність, але вона реалізується через нестандартні додаткові правила, які розширюють базовий лексичний аналіз. У деяких випадках ця система може давати збої, особливо якщо код містить змішані табуляції та пробіли, незвичні стилі форматування або вкладені блоки нетипової структури [22]. Це рідко спричиняє фатальні помилки, але часом потребує тонкого налаштування, що підвищує складність роботи з граматикою.

Важливим є те, що граматика Python для ANTLR підтримується ентузіастами, а не офіційним Python core team. Коли мова оновлюється, спільноті потрібен час, щоб адаптувати граматику. У критичних проєктах це може створити ситуацію, коли нові синтаксичні форми ще не підтримуються, граматика містить тимчасові обмеження або ж потрібне ручне редагування *.g4-файлів. Попри це, граматика стабільна і активно використовується, хоча залежність від спільнотного оновлення може розглядатися як недолік.

Суттєвим недоліком у даному випадку є відсутність повноцінного AST за замовчуванням ANTLR генерує parse tree, а не AST. Parse tree містить усі синтаксичні вузли, включно з технічними деталями, дужками, ключовими словами (наприклад, def, :), які не завжди потрібні в машинному аналізі. У Python-екосистемі більшість існуючих інструментів виробляють спрощене AST, яке безпосередньо відображає логіку коду, а не його синтаксис. Використовуючи ANTLR, доведодиться або будувати AST самостійно або дописувати існуючий метод обходу Visitor, що перетворює parse tree у зручну

структуру. Це збільшує обсяг роботи, хоча й дає значно більше контролю над форматом вихідних даних.

У межах цієї роботи ANTLR відіграє роль модулю аналізу Python-коду. Спочатку завантажується офіційна граматика Python3 для ANTLR (доступна у публічному репозиторії), яка вже містить формальні правила для всіх конструкцій мови: від базових інструкцій і операторів до лямбда-виразів та структур зі складною вкладеністю. Після генерації парсера на Python отриманий код обробляє вихідні файли .py та формує parse tree у вигляді об’єктної структури, з якою можна працювати програмно.

Це особливо важливо для задачі побудови залежностей між файлами. ANTLR дозволяє легко виділяти всі конструкції import/import-from, визначення класів, зв’язки між методами, виклики функцій – завдяки можливості обходу parse tree із custom-візіторами або лісенерами. Таким чином, проєкт може автоматично витягати повну схему залежностей між файлами, формувати граф для GNN-моделі або для візуалізації.

Окрему цінність становить те, що ANTLR повертає дерево із суворо визначеною структурою, де кожен вузол має тип, набір дочірніх елементів та позицію у вихідному коді. Це забезпечує цілковиту відтворюваність та стабільність під час подальшого аналізу. Наприклад, для побудови графа залежностей проєкт обходить дерево й шукає вузли типу import_stmt чи from_stmt, витягуючи інформацію про назви модулів, локальні аліаси, імпорти окремих атрибутів. ANTLR дозволяє зробити це не через пошук регулярними виразами, а через строгий синтаксичний аналіз.

ANTLR забезпечує парсинг коду з високою точністю, оскільки використовує офіційну граматику. Це гарантує, що дерева будуть побудовані коректно навіть для складних конструкцій. Ручне написання парсерів для Python – надзвичайно трудомістка задача, зважаючи на роль відступів, генератори, анонімні функції, контекстні менеджери та інші структури. ANTLR знімає це навантаження.

ANTLR підтримує десятки мов і дозволяє розширювати граматики при зміні вимог. Якщо у межах роботи в майбутньому потрібно буде аналізувати JavaScript, Java або C++, достатньо завантажити відповідну граматику та підключити інший парсер – логіка аналізу лишиться аналогічною. Ця масштабованість робить ANTLR ефективним вибором для систем, що працюють із багатомовними репозиторіями GitHub.

Слід зазначити, що ANTLR може ефективно співпрацювати з моделями штучного інтелекту, адже надає структуроване дерево, яке легко конвертувати у послідовність токенів або у формат, сумісний із sequence-to-sequence моделями на кшталт T5. Наприклад, вузли дерева можна серіалізувати у вигляді (<тип> <зміст>) або у вигляді скорочених токенів, що значно підвищує якість генерації описів коду, оскільки модель отримує формалізовану, не плутану структуру.

ANTLR створює оптимізований парсер, який працює значно швидше, ніж аналоги, створені вручну або засновані на регулярних виразах. Це дозволяє ефективно обробляти великі GitHub-репозиторії, що важливо для побудови повноцінних датасетів і генерації залежностей.

3.3.2. Лінійне представлення AST

Лінійне представлення AST є важливим для навчання і використання моделей типу seq2seq, таких як T5. Воно дозволяє трансформувати складну деревоподібну структуру коду в послідовність токенів або символів, зберігаючи при цьому семантичну інформацію про структуру програми. Це перетворення важливе для інтеграції з моделями обробки природної мови та кодових даних, оскільки стандартні трансформери оперують послідовностями, а не деревами. Існує кілька підходів до лінійного представлення AST, кожен з яких має свої особливості, переваги та обмеження.

Перший підхід ґрунтується на обході AST у глибину (DFS) з використанням префіксного або постфіксного порядку. У цьому випадку лінійна послідовність формується шляхом обходу дерева і додавання вузлів у певному порядку відносно їхніх дітей. Префіксний обхід передбачає запис вузла перед його дітьми, постфіксний – після. Цей метод дозволяє зберігати ієрархічну структуру дерева, оскільки порядок вузлів у послідовності відображає відносини батько-дитина. Для збереження контексту та структури часто використовуються спеціальні токени початку та кінця вузла, що дозволяє моделі чіткіше розуміти межі функцій, операторів і блоків коду. Перевага цього підходу полягає у простоті реалізації та прямій інтеграції з трансформерними моделями, однак при роботі з великими функціями або класами послідовності можуть ставати надто довгими, що ускладнює навчання моделі та потребує додаткових обмежень на максимальну довжину токенів.

Другий популярний метод – це S-expression або Lisp-подібний формат, де AST записується у вигляді вкладених виразів, оточених дужками. Кожен вузол дерева стає підвиразом, а його діти – внутрішніми підвиразами. Такий підхід дозволяє зберегти повну структуру дерева у текстовому вигляді та забезпечує легку зворотну реконструкцію дерева з лінійної послідовності. Він особливо корисний у задачах аналізу коду та генерації коментарів, оскільки моделі отримують інформацію про повні ієрархічні зв’язки між елементами програми. Недоліком є довжина виразу для великих функцій та потреба в додатковій токенізації для трансформерів, оскільки без розділення на окремі токени модель не зможе адекватно опрацьовувати такі рядки.

Третій метод полягає у серіалізації AST у форматі JSON або подібному словниковому представленні. Кожен вузол зберігається як об’єкт із типом, значенням та списком дітей. Цей підхід забезпечує гнучкість, дозволяє легко додавати атрибути вузлів, такі як типи змінних, модулі, коментарі та інші метадані. JSON-подання дуже зручне для програмних маніпуляцій та інтеграції з Python, оскільки його можна безпосередньо обробляти за допомогою стандартних бібліотек. Основна складність полягає в тому, що для

навчання моделей типу T5 або GPT потрібно додатково перетворювати JSON у лінійну послідовність токенів, зберігаючи при цьому структурну інформацію, тому використання даного способу було відкинуто майже одразу.

Наступний підхід – path-based або edge-based представлення, де дерево AST розглядається як набір шляхів від кореня до листя. Кожен шлях фіксує послідовність вузлів, через які проходить шлях від початку дерева до конкретного кінцевого елемента. Така лінійна структура дозволяє скоротити довжину послідовностей і фокусуватися на ключових семантичних відношеннях між елементами коду. Path-based підходи широко використовуються у методах типу Code2Vec та Code2Seq. Вони дають змогу моделі виділяти окремі семантичні контексти, проте при цьому втрачається частина глобальної ієрархічної інформації дерева, що може бути критично для задач повного аналізу коду або генерації коментарів для великих функцій.

П’ятий метод – flattened або tokenized linear AST, який передбачає представлення дерева у вигляді плоскої послідовності токенів із збереженням лише семантично значущих вузлів і операторів. Спеціальні токени позначають початок та кінець ключових блоків коду, що дозволяє трансформеру краще розуміти структуру програми без зберігання всієї деталізації деревоподібної структури. Цей підхід оптимально підходить для інтеграції з моделями типу T5, оскільки вони оперують послідовностями обмеженої довжини, і дозволяє ефективно обробляти великі обсяги коду. Єдиним недоліком є потенційна втрата частини локальної ієрархії, що може впливати на точність генерації детальних коментарів для складних конструкцій.

Порівняльний аналіз цих підходів дозволяє зробити кілька важливих висновків. Preorder/postorder обходи є найбільш прямолінійними і простими для реалізації, але їх довжина може бути критичною для великих проектів. S-expression забезпечує повну структурну інформацію, проте потребує додаткової токенізації. JSON-представлення гнучке і зручне для програмної обробки, але вимагає додаткової лінійної конверсії. Path-based методи скорочують довжини послідовностей і добре працюють для виділення

локальних семантичних контекстів, але частково втрачають глобальну ієрархію. Flattened tokenized AST є компромісом між семантикою та практичністю для навчання трансформерів, зберігаючи основні блоки коду та операції у компактній формі.

У контексті даного проєкту оптимальним рішенням стало використання ANTLR для генерації parse tree із наступною конверсією у flattened linear AST, що дозволяє трансформеру отримувати інформацію про основну структуру коду, мінімізуючи обсяг даних і максимально підвищуючи ефективність навчання. Спеціальні токени для вузлів, таких як FUNCDEF, RETURN, BINOP, VAR, END, дозволяють моделі розрізняти ключові елементи коду та ефективно формувати коментарі. Такий підхід забезпечує баланс між збереженням семантики, компактністю представлення та ефективністю навчання моделі, що робить його оптимальним для генерації документації та автоматичного коментування коду на великих наборах даних.

Таким чином, лінійне представлення AST у вигляді flattened sequences забезпечує високу сумісність із сучасними seq2seq моделями, дозволяє ефективно навчати трансформери на великих обсягах коду, а також зберігає достатню інформацію про структуру програми для якісного автоматичного коментування. Такий формат поєднує простоту реалізації, ефективність навчання та збереження семантичної інформації коду, що забезпечує високу точність генерації коментарів.

3.3.3. Роль AST Node Embeddings у проекті

Абстрактне синтаксичне дерево (AST) є ключовим представленням програмного коду, яке відображає структуру програми у вигляді ієрархії вузлів – операторів, функцій, класів, імпортів, викликів методів тощо. Однак саме по собі AST є символічним, а тому не може бути безпосередньо використане нейронними мережами. Для інтеграції цієї структурної

інформації в GNN необхідно перетворити кожен вузол AST на числовий вектор. Такі вектори називаються AST Node Embeddings.

AST Node Embedding – це компактне векторне представлення вузла AST, яке кодує як його тип (наприклад, FunctionDef, Assign, ClassDef), так і його внутрішній зміст (імена змінних, ключові слова, літерали) та структурні характеристики - глибина у дереві, кількість дочірніх елементів, розмір блоку коду, тощо [32]. Формування таких ембеддингів включає декілька етапів.

Спочатку, за допомогою ANTLR або стандартних засобів Python, вихідний код перетворюється на AST. Кожен вузол дерева класифікується за типом, а тип вузла проходить через вбудований embedding-шар, який навчається подібно до словникових embedding у NLP. Це дозволяє моделі зрозуміти, що певні конструкції мови програмування мають схожі ролі.

Далі з вузла виділяється семантичний зміст: імена змінних, назви функцій, значення літералів або ключові слова. Ці токени обробляються за допомогою токенізатора, після чого їхні векторні представлення агрегуються – наприклад, через середнє або згортку. Таким чином модель отримує інформацію про текстовий зміст вузла.

Окрім цього, формуються числові ознаки, що описують структуру: кількість дочірніх вузлів, довжина тіла функції, кількість параметрів, рівень вкладеності. Це дозволяє врахувати ієрархічні властивості AST. Усі ці компоненти об’єднуються конкатенацією, після чого проходять через проекційний шар, який переводить інформацію в компактний вектор фіксованої розмірност. Такий вектор і є AST Node Embedding.

Отримані ембеддинги виконують подвійну роль у системі. По-перше, вони передаються до GNN, яка аналізує залежності між функціями, файлами або модулями, поширюючи інформацію через структуру графа. Це дозволяє моделі визначити важливі компоненти коду, знайти критичні зв’язки та сформувати узагальнене представлення структури програми.

По-друге, узагальнені графові ембеддинги інтегруються в модель T5 під час генерації коментарів або документації. Завдяки цьому T5 отримує не лише

локальний контекст окремого фрагменту коду, але й розуміння його ролі всередині усього проекту. Такий підхід забезпечує значно більшу точність та інформативність коментарів порівняно з моделями, що працюють виключно з текстом коду.

Таким чином, використання AST Node Embeddings є центральним елементом і основною особливістю всієї розробленої архітектури, бо вони поєднують структурні та семантичні властивості програмного коду у формі, придатній для глибинного навчання, та забезпечують спільну роботу модулів GNN та T5 у рамках запропонованої системи.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному дослідженні було проаналізовано та інтегровано сучасні методи обробки природної мови та програмного коду для задач автоматичного коментування та генерації документації. Основним інструментом для роботи з текстом та кодом виступила модель T5, яка дозволяє уніфіковано формулювати різні завдання NLP у форматі «текст → текст». Завдяки своїй архітектурі, що базується на трансформері, T5 забезпечує ефективне врахування як локальних, так і глобальних залежностей у послідовностях токенів, що критично для аналізу коду, де значення одного елементу часто залежить від контексту всього фрагмента або навіть проекту в цілому.

Використання багатоголової уваги дозволяє моделі одночасно обробляти різні аспекти контексту та виділяти ключові залежності, що підвищує точність прогнозів та генерації коментарів. Функція втрат на основі крос-ентропії забезпечує чітке покарання за невідповідність передбачених та еталонних токенів, що сприяє формуванню точних, послідовних і семантично коректних вихідних текстів.

Процес токенізації за допомогою SentencePiece забезпечує гнучку роботу з будь-яким текстом, включно з кодом, дозволяючи ефективно поділяти його на підпослідовності без втрати структури. Цей підхід дозволяє зменшити кількість рідкісних токенів і забезпечує стабільність навчання, що особливо важливо при роботі з великими наборами коду різних мов та стилів.

Для адаптації моделі T5 під специфічну задачу проєкту було застосовано метод LoRA, який дозволяє ефективно донавчати великі трансформери, додаючи низькоранговані матриці для ключових шарів моделі. Це забезпечує швидку адаптацію до нових доменних даних при мінімальних обчислювальних витратах, зберігаючи при цьому знання, накопичені моделлю під час початкового навчання на великих корпусах. Такий підхід є оптимальним у випадках обмежених ресурсів, оскільки дозволяє швидко

досягати високої якості генерації коментарів без повного навчання усієї моделі.

У межах проекту також була реалізована графова нейронна мережа (GNN) для аналізу залежностей між файлами та модулями коду. Використання GNN дозволяє враховувати складні, нерегулярні зв’язки між компонентами програми, що не можуть бути представлені у вигляді послідовностей або матриць. Багатошарова GAT-модель з механізмом уваги дозволяє диференціювати значущість різних залежностей, забезпечуючи точніше представлення структури проекту. Застосування нормалізації та регуляризації робить процес навчання стабільним і запобігає перенавчанню.

ANTLR виступає ключовим інструментом для формування синтаксичних дерев Python-коду. Генерація parse tree забезпечує формальне та машиночитане представлення структури програми, а лінійне представлення AST у форматі flattened sequences дозволяє інтегрувати ці дані у seq2seq моделі, зберігаючи при цьому основну семантичну інформацію та скорочуючи обсяг даних для навчання. AST Node Embeddings, сформовані на основі дерева, передають структурні та семантичні властивості коду у GNN та T5, забезпечуючи взаємодію між модулями аналізу залежностей та генерації коментарів.

Загалом інтеграція моделей T5 і GNN та ANTLR дозволяє побудувати ефективну систему автоматичного коментування коду, яка поєднує переваги сучасних трансформерів і графових моделей. Такий підхід забезпечує високу точність генерації коментарів, збереження контексту проекту, економію ресурсів при донавчанні і гнучкість у роботі з великими та складними кодовими базами. Високий рівень інтеграції структурних і семантичних характеристик коду дозволяє системі формувати осмислені та інформативні пояснення навіть для великих і складних проектів, що робить її ефективним інструментом для автоматизації документації та підтримки розробки програмного забезпечення.

4. ОПИС ТА ТЕСТУВАННЯ РЕАЛІЗОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1. Структура додатку

Серверна частина проекту є ключовим компонентом, що забезпечує збір, обробку та надання даних для фронт-енду та внутрішніх модулів системи. В даному проекті серверну частину реалізовано з використанням фреймворку Flask, який поєднує простоту розробки з високою гнучкістю та можливістю інтеграції з іншими компонентами Python, такими як PyTorch, Mako, ANTLR та Graph-tool. Основна мета серверно частини додатка - організувати централізоване управління процесом отримання коду з репозиторію, аналізу його структури, генерації коментарів та побудови документації, а також забезпечити взаємодію з користувачем через веб-інтерфейс.

Серверна частина додатку приймає запити від користувацького інтерфейсу, зокрема через REST API або веб-інтерфейс Flask користувачі передають URL публічного GitHub-репозиторію для подальшого аналізу коду, і формування документації. Саме серверна частина ініціює клонування репозиторію, збір вихідних файлів, парсинг за допомогою ANTLR, побудову AST та передачу даних у модель T5 та GNN.

Для підвищення продуктивності та масштабованості використовується бібліотека Celery, який дозволяє виконувати тривалі завдання, такі як донавчання моделі, аналіз великих проектів або генерацію документації у фоновому режимі, не блокуючи основний потік Flask.

Серверна частина управляє процесом генерації коментарів за допомогою донавченої моделі T5, а також аналізу залежностей між модулями проекту за допомогою GNN.

Після обробки коду та генерації коментарів сервер готує структуровані дані, які передаються до шаблонів Mako для динамічної генерації HTML-документації.

У результаті готові дані можуть бути повернуті через API у форматі JSON, HTML-сторінки можуть бути відображені на стороні користувача та доступні для скачування.

Загалом вся система побудована за патерном MVC (Model-View-Controller) і розділена на три шари: модель (внутрішні дані проекту, структури AST, графи залежностей, дані моделей штучного інтелекту), представлення (HTML-сторінки, що генеруються Mako, візуалізація графів через Graph-tool, користувацький інтерфейс React) та контролери (Flask-маршрути, які приймають запити, обробляють їх через модулі аналізу коду, генерації коментарів та підготовки документації).

Архітектура сервера побудована за модульним принципом, що забезпечує легкість масштабування та інтеграції з різними компонентами проекту (рис. 4.1). Якщо кожен компонент виконує окрему функцію, це полегшить підтримку та розширення системи у майутньому. Celery дозволяє виконувати ресурсомісткі завдання паралельно, не блокуючи веб-інтерфейс.

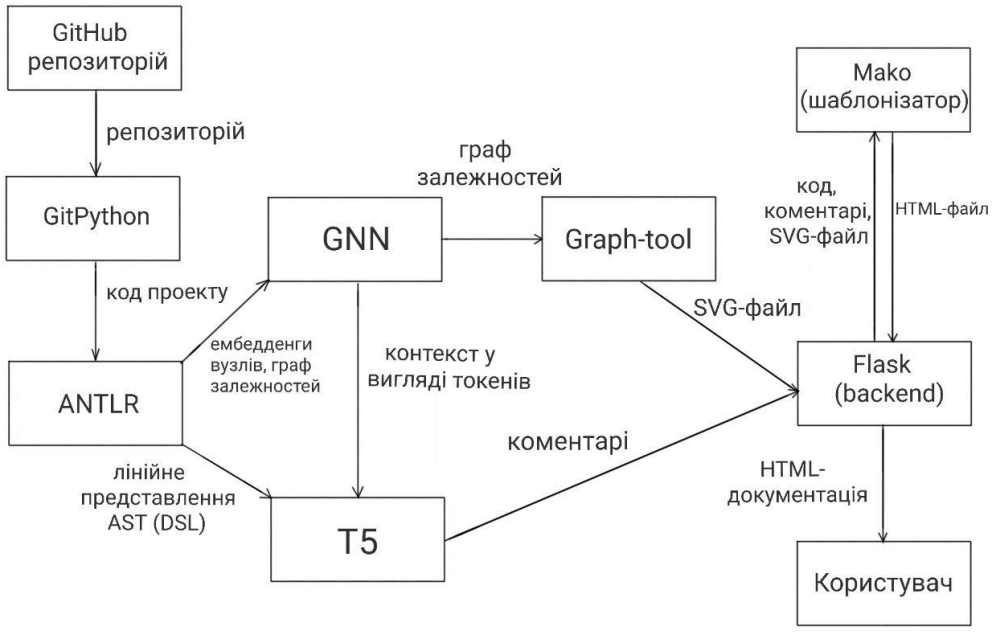

Рисунок 4.1 – Схема взаємодії компонентів додатку

Основними модулями серверної частини є:

1) API-модуль Flask – відповідає за маршрутизацію запитів, валідацію вхідних даних та передачу їх у відповідні внутрішні сервіси. Кожен маршрут Flask може викликати окремі функції обробки, які ініціюють послідовність завдань, пов'язаних із завантаженням репозиторію, аналізом коду та генерацією документації.
2) Модуль роботи з GitHub – реалізує функції клонування репозиторіїв та збирання вихідного коду. Використовується бібліотека GitPython, яка дозволяє зручно керувати локальними копіями репозиторіїв, отримувати список файлів та їх вміст.
3) Модуль парсингу та аналізу коду – відповідає за створення AST за допомогою ANTLR, вилучення функцій, класів, імпортів та експортів. Ці дані використовуються як для створення коментарів T5, так і для побудови графа залежностей GNN.
4) Модуль генерації коментарів T5 – інтерфейс між бек-ендом та моделлю T5. Цей модуль приймає код, перетворений на токени і підготовлений за допомогою SentencePiece, і повертає згенеровані коментарі. Також підтримується донавчання моделі на специфічних датасетах, якщо користувач завантажує власні приклади.
5) Модуль аналізу залежностей GNN – будує графи залежностей між модулями та файлами проекту. Результатом роботи модуля є структуровані графи, які далі візуалізуються за допомогою Graph-tool та інтегруються до документації.
6) Модуль генерації документації Mako — отримує від інших модулів структуровані дані про код, коментарі та графи залежностей і формує багатосторінкову HTML-документацію.
7) Модуль асинхронних завдань Celery — керує тривалими завданнями, такими як обробка великих репозиторіїв, навчання моделі або побудова графів, дозволяючи основному Flask-додатку залишатися чуйним.

Загалом процес обробки даних реалізований у вигляді послідовності взаємопов'язаних етапів. Спочатку користувач надсилає URL-адресу

репозиторію через веб-інтерфейс. Flask перевіряє коректність URL та ініціює завдання клонування репозиторію.

Модуль роботи з GitHub завантажує репозиторій у локальну директорію, після чого збираються всі вихідні файли, що задовольняють заданим розширенням (наприклад, .py для Python), після чого модуль ANTLR будує AST кожного файлу. На основі AST витягуються функції, класи, змінні, імпорти та експорти. Ці дані є структурованим уявленням коду, яке використовується як вхід для T5 і GNN.

Підготовлені дані передаються моделі T5 для генерації коментарів та GNN для побудови графа залежностей. У результаті формуються текстові блоки коментарів та візуальні схеми зв'язків між модулями.

На наступному етапі модуль Mako використовує дані з попередніх кроків для динамічного створення HTML-сторінок. Кожна сторінка містить опис функцій та класів, згенеровані коментарі та візуалізацію залежностей. Готова документація доступна для скачування через користувацький інтерфейс.

Бібліотека Celery забезпечує своєрідний міст між Flask та ресурсозатратними модулями, такими як T5, GNN та Mako. Основний потік Flask не займається обчисленнями - він лише ставить завдання у чергу і відстежує їхній статус. Модуль генерації коментарів T5, аналіз графів GNN та парсинг AST ANTLR отримують дані через Celery та повертають результати до загального пулу бек-енду.

Після того як Flask приймає HTTP-запит від користувача, наприклад, на завантаження репозиторію або генерацію документації, фреймворк ініціює завдання Celery, передаючи всі необхідні параметри, такі як URL репозиторія, шлях для збереження файлів і конфігурації моделей T5 і GNN. Celery ставить завдання у чергу повідомлень, використовуючи брокер (для даного проекту застосовано Redis), який керує послідовністю і розподілом завдань між воркерами.

Воркер Celery забирає завдання з черги і виконує її в окремому процесі, незалежно від основної програми Flask, після чого зберігає результати - згенеровані коментарі, візуалізації графів залежностей або HTML-документацію - і повідомляє інтерфейс користувача про готовність даних. Такий поділ дозволяє уникнути блокування основного веб-потоку, що критично під час роботи з великими репозиторіями або під час навчання моделей, коли процес може займати значний час.

Таким чином, серверн централізовано керує всіма запитами та потоками даних у додатку, забезпечуючи обробку коду, генерацію коментарів, побудову графів залежностей та формування документації. Інтеграція всіх частин додатку дозволяє створювати повноцінну масштабовану систему для аналізу та документування коду, яку можна використовувати в роботі з різними проектами та репозиторіями.

4.2. Отримання коду для аналізу за посиланням

Першим етапом для створення документації є отримання вихідного коду з репозиторію за посиланням. Цей процес є надважливим, оскільки подальший синтаксичний аналіз, побудова графа залежностей та генерація коментарів неможливі без повноцінного доступу до файлової структури проекту.

Для реалізації завантаження коду було обрано модуль GitPython, оскільки він надає високорівневий інтерфейс для клонування репозиторіїв, роботи з гілками та вилучення окремих файлів. GitPython обертає функціональність Git у Python-об'єкти, що дозволяє інтегрувати завантаження репозиторіїв у єдину систему без необхідності прямої взаємодії з командним рядком. Такий підхід підвищує переносимість коду та спрощує його інтеграцію з іншими компонентами проекту.

Процес починається з отримання посилання публічний репозиторій. Користувач вводить URL-репозиторія, наприклад: https://github.com/username/project.git. На цьому етапі система перевіряє

коректність введеного URL, а також перевіряє, чи репозиторій доступний для публічного клонування.

Після успішної валідації виконується клонування репозиторію до локальної директорії проекту. Перед тим, щоб уникнути конфліктів із попередніми версіями репозиторію, перевіряється наявність локальної копії. Якщо вона існує, то система видаляє стару директорію за допомогою стандартних засобів Python (shutil.rmtree). Це забезпечує чистий стан для нового клонування та запобігає накопиченню застарілих файлів, які можуть порушити аналіз

Клонування виконується за допомогою методу Repo.clone_from(repo_url, output_dir), де repo_url – це посилання на GitHub, а output_dir – локальна директорія, в яку будуть збережені всі файли проекту. GitPython автоматично створює структуру папок, повторюючи структуру репозиторію, та зберігає всі файли вихідного коду, конфігурації, документацію та інші ресурси.

Після клонування наступним кроком є пошук та фільтрація файлів вихідного коду. У цій розробці проект орієнтований працювати з файлами мовою Python, проте архітектура модуля дозволяє легко адаптуватися до інших мов, додаючи відповідні розширення файлів.

Для пошуку файлів використовують стандартний обхід дерева каталогів за допомогою функції os.walk(). Цей метод проходить рекурсивно по всіх директоріях репозиторію, збираючи шляхи до файлів, які відповідають заданим розширенням (наприклад, .py для Python). У процесі обходу виключаються несуттєві файли, такі як бінарні файли, тимчасові або файли документації, які не впливають на аналіз коду.

Всі знайдені шляхи зберігаються в масив або список, що дозволяє надалі передавати їх на етап парсингу та побудови AST. Крім того, в системі реалізована можливість зберігати вміст усіх файлів у єдиний файл, що полегшує попередній перегляд проекту та прискорює процес донавчання моделі T5.

У процесі клонування та обробки репозиторію можуть виникати помилки, пов'язані з відсутністю доступу, пошкодженням файлів або некоректним кодуванням. Для забезпечення надійної роботи системи всі операції загорнуті у блоки try-except. У разі помилок система виводить інформативні повідомлення, вказуючи на конкретний файл чи директорію, де сталася помилка.

Особливу увагу приділено обробці кодувань. Усі файли відкриваються з використанням кодування UTF-8 з обробкою помилок (encoding="utf-8" та errors="ignore"), що запобігає збоям за наявності спеціальних символів у вихідному коді.

Ключовими перевагами використання бібліотеки GitPython є можливість роботи з будь-якими публічними репозиторіями GitHub, повна автоматизація процесу і простота інтеграції з іншими складовими системи. Окрім того, такий підхід є гнучким та підтримує різні мови програмування та формати файлів, що може бути корисним у майбутньому.

Додатково, така архітектура дозволяє розширювати функціональність системи, наприклад додавати кешування клонованих репозиторіїв, підтримку приватних репозиторіїв через токени доступу або паралельну обробку декількох репозиторіїв одночасно.

4.3. Генерування файлу документації

4.3.1. Особливості шаблонізатора Mako

Шаблонізатор Mako є інструментом для динамічної генерації тексту, який широко застосовується у програмних системах для створення HTML-документації, конфігураційних файлів, скриптів та іншого коду, що формується автоматично.

Основною особливістю Mako є поєднання синтаксису, схожого на мову Python, з можливістю вставки в шаблон логіки та обробки даних, що дозволяє гнучко керувати процесом генерації вмісту. Це робить Mako оптимальним

вибором для проектів, де необхідно комбінувати статичні частини тексту з динамічною інформацією, отриманою з різних джерел, таких як бази даних, API чи результати аналізу коду.

Однією з ключових особливостей Mako є його шаблонізаційна модель, яка базується на концепції успадкування шаблонів. Це означає, що розробник може створювати базові шаблони, що містять загальну структуру документації, наприклад, заголовки, стилі, навігаційні елементи та футери, а потім наслідувати їх у дочірніх шаблонах, де визначаються специфічні для конкретного модуля чи файлу частини. Такий підхід значно підвищує повторне використання коду, спрощує підтримку та дозволяє централізовано змінювати загальну структуру документації без необхідності редагувати всі окремі файли.

Mako підтримує вбудовану логіку, що включає умовні оператори, цикли, виклики функцій та доступ до змінних контексту, переданих при рендерингу шаблону. Це дає змогу динамічно формувати вміст залежно від отриманих даних: наприклад, при генерації документації для програмного проекту шаблон може автоматично обходити список файлів, витягувати інформацію про функції та класи, їхні параметри та повернуті значення, а потім формувати відповідні блоки опису.

Ще однією важливою особливістю Mako є простота інтеграції з Python. Шаблони Mako компілюються у Python-код, що забезпечує високу продуктивність і дозволяє виконувати складні обчислення або маніпуляції з даними безпосередньо під час рендерингу. Це особливо корисно для проектів, де необхідно обробляти великі обсяги даних або виконувати попередню обробку інформації перед виведенням у формат документації. Наприклад, при автоматичному створенні HTML-документації система може обчислювати кількість викликів функцій, аналізувати структуру AST, формувати графові представлення залежностей модулів, а потім передавати ці дані в шаблон Mako для генерації наочних і зрозумілих описів.

Mako також підтримує фрагментовану генерацію та підключення зовнішніх шаблонів, що дозволяє розбивати документацію на окремі компоненти, а потім збирати їх у єдиний документ або багатосторінкову структуру. Наприклад, для великого Python-проекту можна створити окремі шаблони для опису кожного файлу, класу чи модуля, а потім об’єднати їх у повну HTML-документацію, включаючи індекс, навігаційні посилання та зведені таблиці. Такий підхід підвищує масштабованість системи і дозволяє легко оновлювати лише ті частини документації, які змінилися, без повторного рендерингу всього проекту.

Mako підтримує блоки коду та макроси, що дозволяє створювати повторювані фрагменти документації без дублювання коду. Наприклад, шаблон може містити макрос для формування таблиці з переліком функцій і параметрів, який можна викликати для кожного модуля. Це зручно при генерації великої документації, де одна й та сама структура повторюється багато разів для різних файлів. Використання макросів підвищує підтримуваність системи та робить код шаблонів більш чистим і зрозумілим.

Ще однією перевагою Mako є висока швидкість рендерингу. На відміну від деяких інших шаблонізаторів, Mako компілює шаблони в байткод Python при першому використанні, що дозволяє значно прискорити повторний рендеринг тих самих шаблонів із різними даними. Це особливо важливо у проектах, де документація генерується часто або для великих наборів файлів.

Серед недоліків Mako можна відзначити необхідність розуміння синтаксису Python для створення більш складних шаблонів, а також потенційну складність підтримки великої кількості умовних конструкцій і циклів у шаблоні. Однак ці обмеження компенсуються гнучкістю та продуктивністю, що робить Mako ідеальним вибором для складних систем автоматичного створення документації.

В інтеграції з проектом автоматичного документування коду Mako виступає фінальним етапом, де всі дані, отримані з AST, GNN та трансформерної моделі, перетворюються у кінцевий HTML-документ. Таким

чином, шаблонізатор забезпечує єдиний вихідний формат документації, дозволяє застосовувати стилі CSS, інтерактивні елементи та навігаційні панелі, а також легко масштабувати систему для великих Python-проектів.

4.3.2. Інтеграція шаблонізатора та його роль у системі

Взаємодія з Mako реалізована через передачу структурованих даних AST та результатів GNN у контекст шаблону. Mako отримує словники, списки та інші структури Python, що містять інформацію про файли, функції, класи, їхні параметри, повернуті значення, а також графові властивості вузлів і ребер. На основі цих даних шаблон Mako динамічно формує HTML-документацію: створює блоки для опису кожного файлу, таблиці з переліком функцій та їхніх параметрів, інтерактивні графи, що візуалізують залежності між модулями, і підсумкові секції, що демонструють загальну структуру проекту. Завдяки підтримці умовної логіки та циклів у шаблонах, документація може автоматично адаптуватися до різних типів проектів, враховуючи специфіку структури файлів та розмір проекту.

Однією з переваг використання Mako з AST і GNN є поєднання локального та глобального контекстів. AST забезпечує детальний аналіз на рівні окремих функцій та класів, дозволяючи трансформерній моделі T5 генерувати точні коментарі для конкретних блоків коду. Водночас GNN надає глобальний огляд проекту, вказуючи, як різні модулі взаємодіють між собою та які залежності існують. Mako об’єднує ці два потоки інформації, формуючи документацію, яка одночасно містить детальний опис кожного елементу коду і відображає його значення в контексті всього проекту.

Ще одним важливим аспектом є масштабованість та повторне використання шаблонів. За допомогою функціоналу Mako – макросів та успадкування шаблонів – можна створювати базові шаблони для структурних елементів документації та повторно використовувати їх для різних файлів та модулів. Наприклад, один макрос може формувати таблицю параметрів

функцій, яка викликається для кожного модуля, незалежно від його складності. Це значно полегшує підтримку документації та дозволяє швидко адаптувати систему для нових проектів без необхідності переписування шаблонів.

Інтеграція також передбачає візуалізацію графів залежностей, отриманих від GNN, безпосередньо у документації. Mako підтримує вставку SVG та інтерактивних елементів, що дозволяє використати зображення, отримане за допомогою Graph-tool. Це робить документацію не лише текстовим описом, а й наочним зображенням, що може бути корисно для аналізу та розуміння коду.

4.4. Тестування системи та порівняння результатів

4.4.1. Про метрику BERTScore

BERTScore – це метрика для оцінки якості згенерованого тексту, що використовує попередньо навчені трансформерні моделей, таких як BERT, RoBERTa або їх багатомовні аналоги. На відміну від традиційних метрик на кшталт BLEU або ROUGE, котрі орієнтовані на точний збіг токенів між передбаченим та еталонним текстом, BERTScore враховує семантичну схожість слів та фраз [33]. Це робить метрику більш придатною для завдань генерації природної мови, включаючи коментування коду та рефакторинг документа.

Спочатку і еталонний текст, і згенерований розбиваються на токени, що відповідають обраній моделі трансформера (у даному випадку моделі T5. Для кожного токена обчислюється ембединг за допомогою моделі BERT або її аналога, який можна обрати. Ембединги відображають сенс слова з урахуванням його оточення. Обчислення ембедингу для кожного токена за допомогою трансформера вимагає GPU та великого обсягу пам'яті при великих наборах даних, проте це виправдовується вищою точністю аналізу.

Далі для кожного токена згенерованого тексту знаходиться токен з еталонного тексту, з яким його ембеддинг має найбільшу косинусну схожість (4.1).

$$sim\left(h_i^g, h_j^r\right) = \frac{h_i^g \cdot h_j^r}{\|h_i^g\| \cdot \left\|h_j^r\right\|} \tag{4.1}$$

На основі цих пар обчислюються три показники:

1) P (Precision) (4.2) – наскільки згенерований токен схожий на еталон:

$$P = \frac{1}{|x|} \sum_i \max_j sim\ (x_i, y_j) \tag{4.2}$$

2) R (Recall) (4.3) – наскільки еталонний токен «покритий» створеним токеном:

$$R = \frac{1}{|y|} \sum_j \max_i sim\ (y_j, x_i) \tag{4.3}$$

3) F1 (4.4) – гармонічне середнє значення між R та P:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{4.4}$$

де x_i – токени передбачення, y_j – токени еталону, $sim\ (x_i, y_j)$ – косинусна схожість між векторами.

Під час аналізу враховуються смислові збіги, а не лише ідентичні слова. Наприклад, "create product" та "add a new product" будуть оцінені як близькі за змістом, хоча структура коментаря та окремі слова відрізняються. Більше того, контекстні ембедінги дозволяють відрізняти омоніми та правильно враховувати значення слова у конкретному місці. Проте слід враховувати, що тестування коротких фрагментів можуть давати нестабільні результати. Окрім того, значним недоліком є те, що значення F1 досить точно показує семантичний збіг, проте не пояснює, які конкретні помилки припустилася моделі генерації тексту.

Також передбачена можливість за потреби використовувати різні моделі трансформерного типу для обчислення ембеддингів, у тому числі багатомовні. Це може бути корисно для проектів із кодом та коментарями мовою, відмінною від англійської.

Загалом якість оцінювання залежить від обраної моделі BERT – слабка чи невідповідна модель може знизити точність метрики. Проте дослідження показують, що BERTScore краще корелює з оцінкою якості тексту, виставленої людьми, порівняно з BLEU і ROUGE, що може бути корисним, якщо тестування із залученням людей не передбачене або не є доцільним. У цілому можна стверджувати, що метрика ідеально підходить для автоматизованого тестування якості коментарів та подальшого аналізу результатів експериментів.

4.4.2. Результати тестування системи

Для тестування створеного програмного забезпечення було виокремлено частину даних із набору, який використовувався для навчання моделі. Також було створено власні приклади. Результати проведеного тестування наведено у таблицях 1 та 2.

Таблиця 4.1 – Порівняння результатів різних моделей

Модель	Precision	Recall	F1
T5-base (базова)	0.851	0.857	0.854
mT5-base (багатомовна)	0.863	0.868	0.865
T5 (донавчена на коді з коментарями)	0.894	0.901	0.898

Таблиця 4.2 – Порівняння еталонних та згенерованих коментарів

№	Фрагмент коду	Еталонний коментар	T5-base	mT5-base	T5 (донавчена)	BERTScore F1
1	def add(a, b): return a + b	Returns the sum of two numbers.	Adds numbers together.	Returns sum of values.	Computes and returns the sum of two input numbers.	0.93
2	for i in range(len(items)): process(items[i])	Iterates through the list and processes each element.	Loop over list.	Iterates items in list.	Loops through all items and processes each element.	0.89

№	Фрагмент коду	Еталонний коментар	T5-base	mT5-base	T5 (донавчена)	BERTScore F1
3	if not os.path.exists(path): os.mkdir(path)	Creates directory if it does not exist.	Check and create folder.	Make directory if missing.	Checks if the directory exists, and creates it if necessary.	0.91
4	user_input = input("Enter your name: ")	Reads user input from console.	Gets input.	Takes user text.	Reads a name entered by the user from the console.	0.88
5	try: open(file) except FileNotFoundError: print("Error")	Handles missing file errors gracefully.	File error handling.	Catch missing file.	Tries to open a file and prints an error if it is not found.	0.90

З отриманих результатів можна побачити, що донавчена модель T5, інтегрована з лексичним аналізом та структурними представленнями коду через AST, стабільно перевищує базову T5 та mT5-small у точності генерації коментарів. Зокрема, при середній довжині функції до 30 рядків спостерігалося підвищення F1 приблизно на 4–5% порівняно з базовою моделлю. Це свідчить про ефективність адаптації моделі до доменної задачі та важливість включення структурної інформації.

Загалом базова англомовна модель демонструє задовільні результати на простих текстах, але погано узгоджується з коментарями, що містять терміни програмування. Модель mT5-base завдяки багатомовному переднавчанню модель краще працює із текстами, що містять технічну лексику або синтаксис коду. Проте зберігається певна втрати точності на доменних фразах. Донавчання власної моделі на корпусі з коментарями до коду значно підвищує семантичну узгодженість вихідного тексту з оригінальним коментарем. Модель краще розуміє структурні зв’язки між кодом і природною мовою.

Розроблена модель може коментувати не лише окремі рядки коди, а й визначати основні задачі функцій та давати загальний відгук щодо ролі файлу або модулю у проекті. Додатково було проведено аналіз залежності точності коментарів від кількості рівнів вкладеності програми та довжини згенерованого AST. Отримані результати наведено у таблиці 4.3.

Таблиця 4.3 – Тестування з урахуванням кількості вузлів AST та довжини коду

Файл	Функція / Клас	Коментар до функції	Підсумок файлу	К-сть вузлів AST	Склад-ність	F1
jwt.py	generate_token(user_id)	Generates a JWT token for a specific user	The jwt.py file is responsible for authenticating users via JWT, including generating and validating tokens	35	4	0.87
	verify_token(token)	Checks token validity, signature, and expiration date		20	3	0.85
	get_user_from_token(token)	Extracts the user ID from the token and returns the user data		25	2	0.88
database.py	connect_db()	Establishes a connection to the database using the specified parameters	The database.py file is responsible for managing database connections and executing SQL queries	40	3	0.82
	execute_query(query)	Executes the passed SQL query and returns the result		30	5	0.79
	close_connection()	Closes the active database connection		15	1	0.90
app.py	create_app()	Creates an instance of a Flask application with all the necessary configurations	The app.py file is responsible for initializing and launching the web application, connecting routes and middleware	50	6	0.80
	register_blueprints(app)	Registers all blueprints for project modules		30	3	0.83
	init_extensions(app)	Initializes all additional modules and plugins		35	4	0.85

Аналіз залежності точності від складності програмного коду показав, що при збільшенні кількості умовних операторів, вкладених циклів та взаємозв’язків між модулями точність генерації дещо знижується.

Наприклад, для функцій із складністю понад 10 умовних або циклічних конструкцій F1 зменшувалася порівняно з більш простими функціями. Однак

поєднання AST-представлень та графової моделі GNN дозволяє частково компенсують цю тенденцію, оскільки система зберігає інформацію про глобальні залежності між компонентами коду.

Досліджено також вплив довжини AST на якість генерації. Було встановлено, що для дуже коротких AST (до 15 вузлів) модель часто пропускає деталі коду, генеруючи лише загальні коментарі. Для середньої довжини AST (15–50 вузлів) точність коментарів є найвищою, оскільки модель здатна повністю захопити локальні та глобальні структури коду.

При надзвичайно довгих AST (понад 100 вузлів) спостерігається певне зниження точності, що пов’язано із складністю відстеження всіх зв’язків та обмеженням контексту у трансформерній моделі.

Для зменшення цього ефекту у системі застосовуються додаткові механізми обрізання та пріоритетного аналізу ключових вузлів AST.

ВИСНОВКИ ДО РОЗДІЛУ 4

У даному розділі було детально розглянуто процеси та архітектуру розробленої системи, а також протестовано її та проаналізовано отримані результати.

Серверна частина додатка забезпечує централізоване управління всіма внутрішніми процесами проекту. Модульна архітектура дозволяє легко інтегрувати нові моделі чи візуалізації, замінювати компоненти без зміни загальної структури та масштабувати систему для роботи з великими кодовими базами. Завдяки Celery тривалі завдання виконуються асинхронно, що підвищує стійкість системи. Flask забезпечує гнучкість маршрутизації та легкість взаємодії з користувацьким інтерфейсом, який реалізований з використанням ReactJS.

Крім того, серверна частина відіграє ключову роль у забезпеченні узгодженості даних: усі модулі використовують єдину структуру подання коду, що дозволяє коректно інтегрувати результати аналізу T5 та GNN у документацію Mako.

Під час тестування системи для визначення точності отриманих коментарів було використано метрику BERTScore, яка враховує лексичну та семантичну схожість між еталонним та згенерованим коментарем. Розглянуто коментарі до окремих частин коду, функцій, а також підсумковий коментар до файлу загалом. Також було виявлено залежність точності отриманих коментарів від складності дерева AST та фунціональної складності коду. Результати дослідження показали, що точність згенерованих коментарів за показником F1 обернено пропорційно залежить від об’єму та складності аналізованого коду.

ВИСНОВКИ

У ході виконання роботи було досліджено актуальну проблему генерації опису для коду з метою кращого його розуміння. Об’єктом дослідження був процес генерації опису коду. Предметом дослідження були попередньо навчені моделі обробки природної мови з донавчанням на специфічних наборах даних для виконання задач програмування та документування.

Під час виконання поставлених задач було опрацьовано теоретичний матеріал, проаналізовано вже існуючі для вирішення проблеми сервіси і визначено їхні недоліки. На основі зібраних даних було запропоновано власний підхід покращення якості генерації опису коду з використанням донавченої моделі T5 та створеної моделі GNN з подальшою реалізацією, що і є результатом досліження. Запропонована система поєднує кращі практики синтаксичного аналізу, графового моделювання і трансформерної генерації, забезпечуючи практично застосовне рішення для автоматичного створення документації.

Актуальність проведеного дослідження полягає у сучасних потребах у забезпеченні ефективного управління великими програмними проектами, розробка яких триває протягом тривалого часу та потребує постійного оновлення документації. Автоматизація процесу створення опису коду є особливо важливою, оскільки ручне написання документації є трудомістким, вимагає високого рівня концентрації та глибокого розуміння структури програмних систем. У зв’язку з цим розробка інтегрованих рішень, здатних прискорити та підвищити якість генерації документації, має як наукову, так і практичну значущість.

Основна новизна дослідження полягає у створенні комплексної системи автоматичного документування програмного коду, яка поєднує трансформерну модель T5, донавчену на наборі даних формату «код-коментар», та модель GNN. Для аналізу структури коду використано абстрактні синтаксичні дерева у лінійному представленні. Запропонований

підхід забезпечує врахування контекстів програмного коду, що дозволяє системі точно визначати призначення коду, його внутрішні взаємозв’язки, а отже формувати структуровані коментарі й документацію високої якості.

Експериментальні результати показали ефективність адаптації трансформерної моделі для обраної задачі. Донавчання T5 на спеціалізованому корпусі з прокоментованим кодом у поєднанні з лексичним аналізом підвищило якість генерації приблизно на 4% за метрикою F1 порівняно з базовою моделлю. Це свідчить про те, що цільове донавчання великих трансформерних моделей може суттєво покращити результати автоматичного документування, особливо для великих Python-проектів. Водночас використання GNN для моделювання залежностей між файлами та модулями дозволяє створювати графічні представлення структури проекту, що забезпечує додатковий наочний інструмент для аналізу взаємозв’язків коду.

Наукове значення полягає у демонстрації можливості комбінування різних методів машинного навчання для автоматизації процесу документування коду. Дослідження може слугувати основою для створення нових методів генерації документації, включно з багатомовними системами, інтеграцією з середовищами розробки та автоматизованим контролем якості програмного забезпечення.

Практичне значення роботи полягає у створенні реального інструмента для автоматичної генерації документації, що дозволяє розробникам значно скоротити час на створення описів функцій і модулів, а також ефективно підтримувати великі програмні системи. Поєднання «seq2seq» трансформерів, методу донавчання LoRA, токенізації через SentencePiece, аналізу AST і графового моделювання через GNN забезпечує інтегрований підхід із високою точністю та гнучкістю. Така архітектура дозволяє системі швидко адаптуватися до нових проектів і специфічних завдань, одночасно зберігаючи здатність обробляти як локальні, так і глобальні контексти коду.

Розроблений підхід можна покращити шляхом оптимізації архітектури трансформера під специфічні мови програмування, удосконалення графових моделей для точнішого відображення залежностей між модулями, інтеграцію семантичного аналізу коду та підвищення масштабованості системи для великих корпоративних проектів. Додатково передбачена можливість розробки адаптивних систем, які автоматично оновлюють документацію під час внесення змін у код, що дозволить значно підвищити продуктивність розробників і знизити ризик помилок у великих програмних системах.

Проте слід зазначити, що якість результату багато в чому залежить від даних для донавчання T5 і якості побудованих графів, а продуктивність на дуже великих репозиторіях потребує оптимізації. Також, для деяких мов може знадобитися тонке налаштування парсера та граматики.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Hoffa, J. 400,000 GitHub repositories, 1 billion files, 14 terabytes of code: Spaces or Tabs?. – Електронний ресурс. – URL: https://hoffa.medium.com/400-000-github-repositories-1-billion-files-14-terabytes-of-code-spaces-or-tabs-7cfe0b5dd7fd
2. Wikipedia. Doxygen. Доступно: https://en.wikipedia.org/wiki/Doxygen
3. Doxygen. The Doxygen Architecture. – Електронний ресурс. – URL: https://www.doxygen.nl/manual/arch.html
4. Medium. Using MkDocs to Create Your Documentation. – Електронний ресурс. – URL: https://medium.com/norsys-octogone/using-mkdocs-to-create-your-documentation-48f503219bc4
5. Swagger / OpenAPI Initiative. The OpenAPI Specification. – Електронний ресурс. – URL: https://swagger.io/specification/
6. GitHub, Inc. GitHub Copilot — AI-powered code completion and suggestions. – Електронний ресурс. – URL: https://github.com/features/copilot
7. Swimm. Documentation Generators: Great Tools You Should Know. – Електронний ресурс. – URL: https://swimm.io/learn/documentation-tools/documentation-generators-great-tools-you-should-know
8. Swimm. Documentation Generators: Great Tools You Should Know. – Електронний ресурс. – URL: https://swimm.io/learn/documentation-tools/documentation-generators-great-tools-you-should-know
9. Wikipedia. Tabnine. – Електронний ресурс. – URL: https://en.wikipedia.org/wiki/Tabnine
10. Shiina H., Onishi S., Takahashi A., Kobayashi N. Automatic comment generation for source code using external information by neural networks for computational thinking — International Journal of Smart Computing and Artificial Intelligence. — 2021. — Vol. 5, No. 2. — P. 15–28.
11. Wei B., Li G., Xia X., Fu Q., Jin Z. Retrieve and Refine: Exemplar-based Neural Comment Generation — Proceedings of the 27th International Conference on

Software Analysis, Evolution and Reengineering (SANER). — 2020. — P. 1–12.

12. Krasanakis A., Papadopoulos S., Kompatsiaris I. JGNN: Graph Neural Networks on Native Java — Proceedings of the 20th International Conference on Mining Software Repositories (MSR). — 2023. — P. 1–12.
13. Amazon Web Services. What is GPT?. – Електронний ресурс. – URL: https://aws.amazon.com/what-is/gpt/
14. Medium. How GPT Model Works. – Електронний ресурс. – URL: https://medium.com/@tinparnus/how-gpt-model-work-7ab01d848057
15. Medium. How T5 Works: A Breakdown of the Transformer-based Language Model. – Електронний ресурс. – URL: https://anote-ai.medium.com/how-t5-works-a-breakdown-of-the-transformer-based-language-model-36fcb59c9e3f
16. Wolfe, C. R. T5: Text-to-Text Transformers (Part 1). – Електронний ресурс. – URL: https://cameronrwolfe.substack.com/p/t5-text-to-text-transformers-part
17. Medium. Understanding the T5 Model: A Comprehensive Guide. – Електронний ресурс. – URL: https://medium.com/@gagangupta_82781/understanding-the-t5-model-a-comprehensive-guide-b4d5c02c234b
18. Zhang D., Moussiades L., Ahmad H., et al. Understanding Code Summarization via Graph Neural Networks and Transformers. – arXiv preprint arXiv:2109.01109, 2021. – 17 p.
19. Hugging Face. Transformers Quick Tour. – Електронний ресурс. – URL: https://huggingface.co/docs/transformers/quicktour
20. Abadi M., Barham P., Chen J., et al. TensorFlow: A System for Large-Scale Machine Learning. – 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), 2016. – 19 p.
21. Paszke A., Gross S., Massa F., et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. – Advances in Neural Information Processing Systems (NeurIPS) 2019: Tutorials, 2019. – 12 p.

22. GitHub. ANTLR Documentation. – Електронний ресурс. – URL: https://github.com/antlr/antlr4/blob/master/doc/index.md
23. Wikipedia. Abstract Syntax Tree. – Електронний ресурс. – URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree
24. Clang. LibClang Documentation. – Електронний ресурс. – URL: https://clang.llvm.org/docs/LibClang.html
25. Esprima. Documentation. – Електронний ресурс. – URL: https://esprima.org/doc/
26. Graph-tool. Parallel Algorithms. – Електронний ресурс. – URL: https://graph-tool.skewed.de/static/doc/parallel.html
27. Eschweiler S., Pallickara S. Performance and Scalability of Graph Visualization Libraries: A Comparative Study of Graph-tool and NetworkX. – Journal of Visual Languages & Computing, 2020. – Vol. 54, P. 101–114.
28. The Jinja Software Foundation. Jinja2 — Template Engine for Python. – Електронний ресурс. – URL: https://palletsprojects.com/p/jinja/
29. Mako. Documentation. – Електронний ресурс. – URL: https://makojs.dev/docs/features
30. NEURIPS 2020 Transformers Team. T5: Text-To-Text Transfer Transformer – Official Documentation. – Електронний ресурс. – URL: https://github.com/google-research/text-to-text-transfer-transformer
31. Smith R., Tan L., Rajan H. Graph Neural Networks for Code Analysis: A Comparative Survey. – arXiv preprint arXiv:2307.08932, 2023. – 24 p.
32. Hu S., Li Y., Xia X., Lo D. Reinforcement Learning for Code Summarization with Code Embedding. – Journal of Systems & Software, 2021. – Vol. 172, P. 110898.
33. Zhang T., Kishore V., Wu F. та ін. BERTScore: Evaluating Text Generation with BERT. – Електронний ресурс. – 2020. – URL: https://arxiv.org/abs/1904.09675